Contents

About This Reference  ii
Typographical conventions  liii
Note to Windows users  lv
Technical support  lv

Summary of Functions  1
Summary of function categories  3
Asynchronous I/O functions  6
Atomic functions  7
Character manipulation functions  8
Conversion functions  9
Directory functions  12
Dispatch interface functions  13
File manipulation functions  17
IPC functions  19
Hardware functions  25
Math functions  27
Memory allocation functions  33
Memory manipulation functions  34
Message-queue functions  36
Multibyte character functions  37
QNX Neutrino-specific IPC functions  38
Operating system I/O functions  40
PC Card functions  44
Platform-specific functions  45
Process environment functions 46
Process manipulation functions 50
Realtime timer functions 58
Resource manager functions 60
Searching and sorting functions 66
Shared memory functions 67
Signal functions 67
Stream I/O functions 69
String manipulation functions 73
System database functions 76
System message log functions 77
TCP/IP functions 78
Terminal control functions 86
Thread functions 87
Time functions 96
Variable-length argument list functions 99
Wide-character functions 99
What’s in a function description? 103
Synopsis: 103
Arguments: 104
Library: 104
Description: 104
Returns: 104
Errors: 104
See also: 105
Examples: 105
Classification: 105
Function safety: 109

**Manifests** 111

**QNX Neutrino Functions and Macros** 115

* abort() 119
* abs() 121
accept() 123
access() 126
acos(), acosf() 129
acosh(), acoshf() 131
addrinfo 133
aio_cancel() 135
aio_error() 137
aio_fsync() 139
aio_read() 141
aio_return() 143
aio_suspend() 145
aio_write() 147
alarm() 149
alloca() 152
alphasort() 155
_ambblksz 157
_argc 159
_argv 160
asctime(), asctime_r() 161
asin(), asinf() 164
asinh(), asinhf() 166
assert() 168
asyncmsg_channel_create() 171
asyncmsg_channel_destroy() 174
asyncmsg_connect_attach() 176
asyncmsg_connect_attr() 179
asyncmsg_connect_detach() 181
_asyncmsg_connection_attr 183
asyncmsg_flush() 185
asyncmsg_free() 187
asyncmsg_get() 189
asyncmsg_malloc() 191
asyncmsg_put(), asyncmsg_putv() 193
atan(), atanf() 196
atan2(), atan2f() 198
atanh(), atanhf() 200
atexit() 202
atof() 205
atoi() 207
atoi() 209
atol(), atoll() 211
atomic_add() 213
atomic_add_value() 215
atomic_clr() 217
atomic_clr_value() 219
atomic_set() 221
atomic_set_value() 223
atomic_sub() 225
atomic_sub_value() 227
atomic_toggle() 229
atomic_toggle_value() 231
_auxv 233
basename() 234
bcmp() 237
bcopy() 239
bind() 241
bindresvport() 244
brk() 246
bsearch() 249
_btext 252
btowc() 253
bzero() 255
cabs(), cabsf() 257
cache_fini() 259
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE_FLUSH()</td>
<td>261</td>
</tr>
<tr>
<td>cache_init()</td>
<td>264</td>
</tr>
<tr>
<td>CACHE_INVALID()</td>
<td>270</td>
</tr>
<tr>
<td>calloc()</td>
<td>273</td>
</tr>
<tr>
<td>cbrt(), cbrtf()</td>
<td>275</td>
</tr>
<tr>
<td>ceil(), ceilf()</td>
<td>277</td>
</tr>
<tr>
<td>cfgetispeed()</td>
<td>279</td>
</tr>
<tr>
<td>cfgetospeed()</td>
<td>282</td>
</tr>
<tr>
<td>cfopen()</td>
<td>285</td>
</tr>
<tr>
<td>cfmakeraw()</td>
<td>290</td>
</tr>
<tr>
<td>cfree()</td>
<td>292</td>
</tr>
<tr>
<td>cfsetispeed()</td>
<td>294</td>
</tr>
<tr>
<td>cfsetospeed()</td>
<td>297</td>
</tr>
<tr>
<td>ChannelCreate(), ChannelCreate_r()</td>
<td>300</td>
</tr>
<tr>
<td>ChannelDestroy(), ChannelDestroy_r()</td>
<td>307</td>
</tr>
<tr>
<td>chdir()</td>
<td>309</td>
</tr>
<tr>
<td>chmod()</td>
<td>312</td>
</tr>
<tr>
<td>chown()</td>
<td>316</td>
</tr>
<tr>
<td>chroot()</td>
<td>319</td>
</tr>
<tr>
<td>chsize()</td>
<td>322</td>
</tr>
<tr>
<td>clearemenv()</td>
<td>325</td>
</tr>
<tr>
<td>clearerr()</td>
<td>328</td>
</tr>
<tr>
<td>clock()</td>
<td>330</td>
</tr>
<tr>
<td>clock_gettime()</td>
<td>336</td>
</tr>
<tr>
<td>clock_getres()</td>
<td>334</td>
</tr>
<tr>
<td>clock_getcpuclockid()</td>
<td>332</td>
</tr>
<tr>
<td>clock_nanosleep()</td>
<td>339</td>
</tr>
<tr>
<td>clock_settime()</td>
<td>343</td>
</tr>
<tr>
<td>ClockAdjust(), ClockAdjust_r()</td>
<td>346</td>
</tr>
<tr>
<td>ClockCycles()</td>
<td>349</td>
</tr>
<tr>
<td>ClockId(), ClockId_r()</td>
<td>351</td>
</tr>
<tr>
<td>ClockPeriod(), ClockPeriod_r()</td>
<td>354</td>
</tr>
</tbody>
</table>
ClockTime(), ClockTime_r() 358
close() 361
closedir() 363
closelog() 366
_cmfld() 367
_cmsdname() 368
confstr() 371
connect() 376
ConnectAttach(), ConnectAttach_r() 379
ConnectClientInfo(), ConnectClientInfo_r() 384
ConnectDetach(), ConnectDetach_r() 388
ConnectFlags(), ConnectFlags_r() 390
ConnectServerInfo(), ConnectServerInfo_r() 393
copysign(), copysignf() 396
cos(), cosf() 398
cosh(), coshf() 400
creat(), creat64() 402
crypt() 406
ctermid() 408
ctime(), ctime_r() 410
daemon() 413
daylight 415
DebugBreak() 416
DebugKDBreak() 418
DebugKDOutput() 419
delay() 421
devctl() 423
diff() 433
dirctcl() 435
dirname() 438
dispatch_block() 441
dispatch_context_alloc() 445
dispatch_context_free() 448
dispatch_create() 450
dispatch_destroy() 453
dispatch_handler() 456
dispatch_timeout() 460
dispatch_unblock() 463
div() 465
diaddr() 467
dlclose() 470
dlerror() 472
dlopen() 474
dlsym() 481
dn_comp() 484
dn_expand() 486
drand48() 488
drem(), dremf() 490
ds_clear() 492
ds_create() 494
ds_deregister() 497
ds_flags() 499
ds_get() 501
ds_register() 503
ds_set() 505
dup() 507
dup2() 510
eaccess() 513
_edata 516
encrypt() 517
_end 519
endgrent() 520
endhostent() 521
ENDIAN_BE16() 522
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDIAN_BE32()</td>
<td>524</td>
</tr>
<tr>
<td>ENDIAN_BE64()</td>
<td>526</td>
</tr>
<tr>
<td>ENDIAN_LE16()</td>
<td>528</td>
</tr>
<tr>
<td>ENDIAN_LE32()</td>
<td>530</td>
</tr>
<tr>
<td>ENDIAN_LE64()</td>
<td>532</td>
</tr>
<tr>
<td>ENDIAN_RET16()</td>
<td>534</td>
</tr>
<tr>
<td>ENDIAN_RET32()</td>
<td>536</td>
</tr>
<tr>
<td>ENDIAN_RET64()</td>
<td>538</td>
</tr>
<tr>
<td>ENDIAN_SWAP16()</td>
<td>540</td>
</tr>
<tr>
<td>ENDIAN_SWAP32()</td>
<td>542</td>
</tr>
<tr>
<td>ENDIAN_SWAP64()</td>
<td>544</td>
</tr>
<tr>
<td>endnetent()</td>
<td>546</td>
</tr>
<tr>
<td>endprotoent()</td>
<td>547</td>
</tr>
<tr>
<td>endpwent()</td>
<td>548</td>
</tr>
<tr>
<td>endservent()</td>
<td>550</td>
</tr>
<tr>
<td>endspent()</td>
<td>551</td>
</tr>
<tr>
<td>endutent()</td>
<td>553</td>
</tr>
<tr>
<td>environ</td>
<td>555</td>
</tr>
<tr>
<td>eof()</td>
<td>556</td>
</tr>
<tr>
<td>erand48()</td>
<td>559</td>
</tr>
<tr>
<td>erf(), erff()</td>
<td>561</td>
</tr>
<tr>
<td>erfc(), erfcf()</td>
<td>563</td>
</tr>
<tr>
<td>err(), errx()</td>
<td>565</td>
</tr>
<tr>
<td>errno</td>
<td>568</td>
</tr>
<tr>
<td>_etext</td>
<td>576</td>
</tr>
<tr>
<td>execcl()</td>
<td>577</td>
</tr>
<tr>
<td>execle()</td>
<td>583</td>
</tr>
<tr>
<td>execlp()</td>
<td>590</td>
</tr>
<tr>
<td>execlpe()</td>
<td>596</td>
</tr>
<tr>
<td>execv()</td>
<td>601</td>
</tr>
<tr>
<td>execve()</td>
<td>607</td>
</tr>
<tr>
<td>execvp()</td>
<td>613</td>
</tr>
</tbody>
</table>
execvp() 619
_exit() 624
exit() 627
exp(), expf() 630
expm1(), expm1f() 632

C Library — F to H 635
fabs(), fabsf() 638
fcfgopen() 640
fchmod() 642
fchown() 645
fclose() 648
fcloseall() 650
fchown() 645
fcntl() 652
fdatasync() 661
fdopen() 663
feof() 666
ferror() 668
fflush() 670
fflush() 670
fgetc() 673
fgetchar() 675
fgetpos() 677
fgets() 679
fgetspent() 682
fgetwc() 685
fgetws() 687
fileno() 690
finite(), finitef() 693
flink() 695
flock() 698
flockfile() 701
floor(), floorf() 703
flushall()  705  
fmod(), fmodf()  707  
fnmatch()  710  
fopen(), fopen64()  714  
fork()  719  
forkpty()  723  
fp_exception_mask()  725  
fp_exception_value()  728  
fp_precision()  731  
fp_rounding()  734  
pathconf()  736  
printf()  739  
putc()  741  
putchar()  743  
puts()  745  
putwc()  747  
putws()  749  
read()  751  
free()  754  
freeaddrinfo()  756  
freeifaddrs()  758  
freopen(), freopen64()  760  
frexp(), frexpf()  764  
scanf()  766  
seek(), fseeko()  768  
tfoot()  771  
fstat(), fstat64()  773  
fstatvfs(), fstatvfs64()  777  
fsync()  781  
ftell(), ftello()  783  
ftime()  786  
ftunamate(), ftunamate64()  789
ftrylockfile()  792
ftw(), ftw64()  794
funlockfile()  797
futime()  799
fwrite()  802
fwrite()  804
fwrite()  806
fwsprintf()  809
gai_strerror()  811
gamma(), gamma_r(), gammaf(), gammaf_r()  813
getaddrinfo()  816
getc()  823
getc_unlocked()  825
getchar()  827
getchar_unlocked()  829
cwd()  831
getdtablesize()  836
getegid()  838
getenv()  840
geteuid()  842
getgids()  844
getgrent()  846
getgrgid()  849
getgrgid_r()  851
getgrnam()  854
getgrnam_r()  856
getgrouplist()  859
getgroups()  862
gethostbyaddr()  864
gethostbyaddr_r()  867
gethostbyname(), gethostbyname2()  870
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gethostbyname_r()</td>
<td>873</td>
</tr>
<tr>
<td>gethostent()</td>
<td>876</td>
</tr>
<tr>
<td>gethostent_r()</td>
<td>878</td>
</tr>
<tr>
<td>gethostname()</td>
<td>881</td>
</tr>
<tr>
<td>getifaddrs()</td>
<td>883</td>
</tr>
<tr>
<td>GETIOVBASE()</td>
<td>885</td>
</tr>
<tr>
<td>GETIOVLEN()</td>
<td>887</td>
</tr>
<tr>
<td>getitimer()</td>
<td>889</td>
</tr>
<tr>
<td>getlogin()</td>
<td>891</td>
</tr>
<tr>
<td>getlogin_r()</td>
<td>893</td>
</tr>
<tr>
<td>getnameinfo()</td>
<td>895</td>
</tr>
<tr>
<td>getnetbyaddr()</td>
<td>900</td>
</tr>
<tr>
<td>getnetbyname()</td>
<td>902</td>
</tr>
<tr>
<td>getnetent()</td>
<td>904</td>
</tr>
<tr>
<td>getopt()</td>
<td>906</td>
</tr>
<tr>
<td>getpass()</td>
<td>912</td>
</tr>
<tr>
<td>getpeernam()</td>
<td>914</td>
</tr>
<tr>
<td>getpgid()</td>
<td>916</td>
</tr>
<tr>
<td>getpgroup()</td>
<td>918</td>
</tr>
<tr>
<td>getpid()</td>
<td>920</td>
</tr>
<tr>
<td>getppid()</td>
<td>922</td>
</tr>
<tr>
<td>getprio()</td>
<td>924</td>
</tr>
<tr>
<td>getprotobyname()</td>
<td>926</td>
</tr>
<tr>
<td>getprotobynumber()</td>
<td>928</td>
</tr>
<tr>
<td>getprotoent()</td>
<td>930</td>
</tr>
<tr>
<td>getpwent()</td>
<td>932</td>
</tr>
<tr>
<td>getpwnam()</td>
<td>935</td>
</tr>
<tr>
<td>getpwnam_r()</td>
<td>937</td>
</tr>
<tr>
<td>getpwuid()</td>
<td>940</td>
</tr>
<tr>
<td>getpwuid_r()</td>
<td>942</td>
</tr>
<tr>
<td>getrusage()</td>
<td>945</td>
</tr>
<tr>
<td>getrusage64()</td>
<td>949</td>
</tr>
</tbody>
</table>
gets() 954
getservbyname() 956
getservbyport() 958
getservent() 960
getsid() 962
getsockname() 964
getsockopt() 966
getspent(), getspent_r() 977
getspnam(), getspnam_r() 981
getsubopt() 984
gmtimeofday() 989
gmtime() 991
getutent() 993
getutid() 995
getutline() 998
getw() 1001
getwc() 1003
getwchar() 1005
getwd() 1007
glob() 1009
globfree() 1014
gmtime() 1016
gmtime_r() 1018
h_errno 1020
hcreate() 1022
hdestroy() 1024
herror() 1026
hostent 1029
hsearch() 1031
hstrerror() 1035
htonl() 1037
htons() 1039
C Library — I to L 1051

ICMP 1054
ICMP6 1056
if_freenamindex() 1059
if_indextoname() 1061
if_nameindex() 1063
if_nametoindex() 1065
ifaddrs 1067
ilogb(), ilogbf() 1069
in8() 1071
in8s() 1073
in16(), inbe16(), inle16() 1075
in16s() 1077
in32(), inbe32(), inle32() 1079
in32s() 1081
index() 1083
inet_addr() 1085
inet_aton() 1087
inet_inaof() 1089
inet_makeaddr() 1091
inet_net_ntop() 1093
inet_netof() 1096
inet_netpton() 1098
inet_network() 1100
inet_ntoa() 1102
inet_ntoa_r() 1104
inet_ntop() 1106
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inet_pton()</code></td>
<td>1109</td>
</tr>
<tr>
<td><code>INET6</code></td>
<td>1114</td>
</tr>
<tr>
<td><code>inet6_option_alloc()</code></td>
<td>1118</td>
</tr>
<tr>
<td><code>inet6_option_append()</code></td>
<td>1120</td>
</tr>
<tr>
<td><code>inet6_option_find()</code></td>
<td>1122</td>
</tr>
<tr>
<td><code>inet6_option_init()</code></td>
<td>1124</td>
</tr>
<tr>
<td><code>inet6_option_next()</code></td>
<td>1126</td>
</tr>
<tr>
<td><code>inet6_option_space()</code></td>
<td>1128</td>
</tr>
<tr>
<td><code>inet6_rthdr_add()</code></td>
<td>1130</td>
</tr>
<tr>
<td><code>inet6_rthdr_getaddr()</code></td>
<td>1132</td>
</tr>
<tr>
<td><code>inet6_rthdr_getflags()</code></td>
<td>1134</td>
</tr>
<tr>
<td><code>inet6_rthdr_init()</code></td>
<td>1136</td>
</tr>
<tr>
<td><code>inet6_rthdr_lasthop()</code></td>
<td>1138</td>
</tr>
<tr>
<td><code>inet6_rthdr_reverse()</code></td>
<td>1140</td>
</tr>
<tr>
<td><code>inet6_rthdr_segments()</code></td>
<td>1142</td>
</tr>
<tr>
<td><code>inet6_rthdr_space()</code></td>
<td>1144</td>
</tr>
<tr>
<td><code>initgroups()</code></td>
<td>1146</td>
</tr>
<tr>
<td><code>initstate()</code></td>
<td>1148</td>
</tr>
<tr>
<td><code>input_line()</code></td>
<td>1151</td>
</tr>
<tr>
<td><code>InterruptAttach()</code>, <code>InterruptAttach_r()</code></td>
<td>1154</td>
</tr>
<tr>
<td><code>InterruptAttachEvent()</code>, <code>InterruptAttachEvent_r()</code></td>
<td>1163</td>
</tr>
<tr>
<td><code>InterruptDetach()</code>, <code>InterruptDetach_r()</code></td>
<td>1169</td>
</tr>
<tr>
<td><code>InterruptDisable()</code></td>
<td>1172</td>
</tr>
<tr>
<td><code>InterruptEnable()</code></td>
<td>1174</td>
</tr>
<tr>
<td><code>InterruptHookIdle()</code></td>
<td>1176</td>
</tr>
<tr>
<td><code>InterruptHookTrace()</code></td>
<td>1180</td>
</tr>
<tr>
<td><code>InterruptLock()</code></td>
<td>1182</td>
</tr>
<tr>
<td><code>InterruptMask()</code></td>
<td>1184</td>
</tr>
<tr>
<td><code>InterruptUnlock()</code></td>
<td>1187</td>
</tr>
<tr>
<td><code>InterruptUnmask()</code></td>
<td>1189</td>
</tr>
<tr>
<td><code>InterruptWait()</code>, <code>InterruptWait_r()</code></td>
<td>1191</td>
</tr>
<tr>
<td><code>_intr_v86()</code></td>
<td>1194</td>
</tr>
</tbody>
</table>
iofunc_lseek_default() 1286
iofunc_mknod() 1288
iofunc_mmap() 1291
iofunc_mmap_default() 1295
iofunc_notify() 1297
iofunc_notify_remove() 1304
iofunc_notify_trigger() 1306
iofunc_ocb_attach() 1309
iofunc_ocb_alloc() 1311
iofunc_ocb_detach() 1314
iofunc_ocb_free() 1317
iofunc_ocb_t 1319
iofunc_open() 1322
iofunc_open_default() 1327
iofunc_openfd() 1330
iofunc_openfd_default() 1334
iofunc_pathconf() 1336
iofunc_pathconf_default() 1339
iofunc_read_default() 1341
iofunc_read_verify() 1343
iofunc_readlink() 1347
iofunc_rename() 1350
iofunc_space_verify() 1354
iofunc_stat() 1358
iofunc_stat_default() 1360
iofunc_sync() 1363
iofunc_sync_default() 1365
iofunc_sync_verify() 1367
iofunc_time_update() 1370
iofunc_unblock() 1372
iofunc_unblock_default() 1374
iofunc_unlink() 1377
iofunc_unlock_ocb_default() 1380
iofunc_utime() 1382
iofunc_utime_default() 1385
iofunc_write_default() 1388
iofunc_write_verify() 1390
ionotify() 1394

IP  1400
IPsec  1406
ipsec_dump_policy() 1413
ipsec_get_policyle() 1415
ipsec_set_policy() 1417
ipsec_strerror() 1421

IP6  1423
isalnum() 1433
isalpha() 1435
isascii() 1437
isatty() 1439
iscntrl() 1441
isdigit() 1443
isfdtype() 1445
isgraph() 1447
isin(), isninf() 1449
islower() 1451
isnan(), isnanf() 1453
isprint() 1455
ispunct() 1457
isspace() 1459
isupper() 1462
iswalnum() 1464
iswalpha() 1466
iswcntrl() 1468
iswctype() 1470
iswdigit() 1472
iswgraph() 1474
iswlower() 1476
iswprint() 1478
iswpunct() 1480
iswspace() 1482
iswupper() 1484
iswxdigit() 1486
isxdigit() 1488
itoa() 1490
j0(), j0f() 1493
j1(), j1f() 1495
jn(), jnf() 1497
jrand48() 1499
kill() 1501
killpg() 1504
labs() 1506
lchown() 1508
lcong48() 1511
ldexp(), ldexpf() 1513
ldiv() 1515
lfind() 1517
lgamma(), lgamma_r(), lgammaf(), lgammaf_r() 1520
link() 1523
l io_listio() 1527
listen() 1532
localeconv() 1534
localtime() 1539
localtime_r() 1541
lockf() 1543
log(), logf() 1547
log1p(), log1pf() 1549
log10(), log10f() 1551
logb(), logbf() 1553
login_tty() 1556
longjmp() 1558
lrand48() 1561
lsearch() 1563
lseek(), lseek64() 1566
lstat(), lstat64() 1570
ltoa(), lltoa() 1573
ltrunc() 1576

C Library — M to O 1581
main() 1584
mallinfo() 1587
malloc() 1589
mallopt() 1591
max() 1595
mblen() 1597
mbrlen() 1600
mbtowc() 1602
mbsinit() 1605
mbstowcs() 1607
mbstowcs() 1609
mbtowc() 1612
mcheck() 1615
mem_offset(), mem_offset64() 1617
memalign() 1621
memccpy() 1623
memchr() 1625
memcmp() 1627
memcpy() 1629
memcpyv() 1631
memcpy() 1633
memmove() 1635
memset() 1637
message_attach() 1639
message_connect() 1646
message_detach() 1649
min() 1652
mkdir() 1654
mkfifo() 1657
mknod() 1660
mkstemp() 1664
mktemp() 1666
mktime() 1668
mlock() 1671
mlockall() 1674
mmap(), mmap64() 1678
mmap_device_io() 1685
mmap_device_memory() 1687
modem_open() 1691
modem_read() 1696
modem_script() 1700
modem_write() 1708
modf(), modff() 1711
mount() 1713
mount_parse_generic_args() 1716
mprobe() 1720
mprotect() 1723
mq_close() 1726
mq_getattr() 1728
mq_notify() 1731
mq_open() 1734
mq_receive() 1739
mq_send() 1742
mq_setattr() 1745
mq_timedreceive() 1748
mq_timedsend() 1751
mq_unlink() 1754
mrand48() 1757
__msg_info 1759
MsgDeliverEvent(),(MsgDeliverEvent_r) 1762
MsgError(),(MsgError_r) 1770
MsgInfo(),(MsgInfo_r) 1773
MsgKeyData(),(MsgKeyData_r) 1775
MsgRead(),(MsgRead_r) 1783
MsgReadv(),(MsgReadv_r) 1787
MsgReceive(),(MsgReceive_r) 1791
MsgReceivePulse(),(MsgReceivePulse_r) 1797
MsgReceivePulsev(),(MsgReceivePulsev_r) 1801
MsgReceivev(),(MsgReceivev_r) 1805
MsgReply(),(MsgReply_r) 1809
MsgReplyv(),(MsgReplyv_r) 1812
MsgSend(),(MsgSend_r) 1816
MsgSendnc(),(MsgSendnc_r) 1821
MsgSendPulse(),(MsgSendPulse_r) 1825
MsgSendsv(),(MsgSendsv_r) 1830
MsgSendsvnc(),(MsgSendsvnc_r) 1834
MsgSendv(),(MsgSendv_r) 1838
MsgSendvnc(),(MsgSendvnc_r) 1842
MsgSendvs(),(MsgSendvs_r) 1846
MsgSendvssnc(),(MsgSendvssnc_r) 1850
MsgVerifyEvent(),(MsgVerifyEvent_r) 1854
MsgWrite(),(MsgWrite_r) 1856
MsgWriterv(),(MsgWriterv_r) 1860
msync() 1863
munlock() 1867
munlockall() 1869
munmap() 1871
munmap_device_io() 1873
munmap_device_memory() 1875
munmap_flags() 1877
name_attach() 1880
name_close() 1887
name_detach() 1889
name_open() 1891
nanosleep() 1894
nanospin() 1896
nanospin_calibrate() 1898
nanospin_count() 1901
nanospin_ns() 1903
nanospin_ns_to_count() 1905
nap() 1908
napms() 1909
nbacconnect() 1910
nbacconnect_result() 1913
ND_NODE_CMP() 1915
netent 1917
netmgr_ntostr() 1918
netmgr_remote_nd() 1923
netmgr_strtond() 1925
nextafter(), nextafterf() 1927
nftw(), nftw64() 1930
nice() 1934
nrand48() 1936
nsec2timespec() 1938
ntohl() 1940
ntohs() 1942
offsetof() 1944
open(), open64() 1946
opendir() 1954
openfd() 1957
openlog() 1960
openpty() 1963
out8() 1965
out8s() 1967
out16(), outbe16(), outle16() 1969
out16s() 1971
out32(), outbe32(), outle32() 1973
out32s() 1975

C Library — P to R 1977
pathconf() 1980
pathfind(), pathfind_r() 1983
pathmgr_symmlink() 1987
pathmgr_unlink() 1989
pause() 1991
pccard_arm() 1993
pccard_attach() 1997
pccard_detach() 1999
pccard_info() 2001
pccard_lock() 2004
pccard_raw_read() 2006
pccard_unlock() 2008
pci_attach() 2010
pci_attach_device() 2012
pci_detach() 2022
pci_detach_device() 2024
pci_find_class() 2026
pci_find_device() 2029
pci_irq_routing_options() 2032
pci_map_irq() 2035
pci_present() 2037
pci_read_config() 2040
pci_read_config8() 2043
pci_read_config16() 2045
pci_read_config32() 2048
pci_rescan_bus() 2050
pci_write_config() 2052
pci_write_config8() 2055
pci_write_config16() 2058
pci_write_config32() 2061
pclose() 2063
perror() 2065
pipe() 2067
poll() 2069
popen() 2076
posix_mem_offset(), posix_mem_offset64() 2080
posix_memalign() 2083
posix_TYPED_MEM_GET_INFO() 2085
posix_TYPED_MEM_OPEN() 2088
pow(), powf() 2092
pread(), pread64() 2094
printf() 2096
procmgr_daemon() 2106
procmgr_event_notify() 2108
procmgr_event_trigger() 2113
procmgr_guardian() 2115
procmgr_session() 2119
__progname 2122
protoent 2123
pthread_abort() 2124
pthread_atfork() 2126
pthread_attr_destroy() 2128
pthread_attr_getdetachstate()  2130
pthread_attr_getguardsize()  2132
pthread_attr_getinheritsched()  2134
pthread_attr_getschedparam()  2136
pthread_attr_getschedpolicy()  2138
pthread_attr_getscope()  2140
pthread_attr_getstackaddr()  2142
pthread_attr_getstacklazy()  2144
pthread_attr_getstackprealloc()  2146
pthread_attr_getstacksize()  2148
pthread_attr_init()  2150
pthread_attr_setdetachstate()  2153
pthread_attr_setguardsize()  2155
pthread_attr_setinheritsched()  2157
pthread_attr_setschedparam()  2159
pthread_attr_setschedpolicy()  2161
pthread_attr_setscope()  2163
pthread_attr_setstackaddr()  2165
pthread_attr_setstacklazy()  2167
pthread_attr_setstackprealloc()  2169
pthread_attr_setstacksize()  2171
pthread_barrier_destroy()  2173
pthread_barrier_init()  2175
pthread_barrier_wait()  2177
pthread_barrierattr_destroy()  2179
pthread_barrierattr_getpshared()  2181
pthread_barrierattr_init()  2183
pthread_barrierattr_setpshared()  2185
pthread_cancel()  2187
pthread_cleanup_pop()  2189
pthread_cleanup_push()  2191
pthread_cond_broadcast()  2194
pthread_cond_destroy() 2196
pthread_cond_init() 2198
pthread_cond_signal() 2200
pthread_cond_timedwait() 2202
pthread_cond_wait() 2206
pthread_condattr_destroy() 2210
pthread_condattr_getclock() 2212
pthread_condattr_getpshared() 2214
pthread_condattr_init() 2216
pthread_condattr_setclock() 2218
pthread_condattr_setpshared() 2220
pthread_create() 2222
pthread_detach() 2227
pthread_equal() 2229
pthread_exit() 2231
pthread_getconcurrency() 2233
pthread_getcpuclockid() 2235
pthread_getname_np() 2237
pthread_getschedparam() 2239
pthread_getspecific() 2241
pthread_join() 2243
pthread_key_create() 2245
pthread_key_delete() 2249
pthread_kill() 2251
pthread_mutex_destroy() 2253
pthread_mutex_getprioceiling() 2255
pthread_mutex_init() 2257
pthread_mutex_lock() 2259
pthread_mutex_setprioceiling() 2263
pthread_mutex_timedlock() 2265
pthread_mutex_trylock() 2268
pthread_mutex_unlock() 2270
pthread_mutexattr_destroy() 2272
pthread_mutexattr_getprioceiling() 2274
pthread_mutexattr_getprotocol() 2276
pthread_mutexattr_getpshared() 2279
pthread_mutexattr_getrecursive() 2281
pthread_mutexattr_gettype() 2283
pthread_mutexattr_init() 2286
pthread_mutexattr_setprioceiling() 2289
pthread_mutexattr_setprotocol() 2291
pthread_mutexattr_setpshared() 2293
pthread_mutexattr_setrecursive() 2295
pthread_mutexattr_settype() 2297
pthread_once() 2300
pthread_rwlock_destroy() 2303
pthread_rwlock_init() 2305
pthread_rwlock_rdlock() 2308
pthread_rwlock_timedrdlock() 2310
pthread_rwlock_timedwrlock() 2313
pthread_rwlock_tryrdlock() 2316
pthread_rwlock_trywrlock() 2318
pthread_rwlock_unlock() 2320
pthread_rwlock_wrlock() 2322
pthread_rwlockattr_destroy() 2324
pthread_rwlockattr_getpshared() 2326
pthread_rwlockattr_init() 2328
pthread_rwlockattr_setpshared() 2330
pthread_self() 2332
pthread_setcancelstate() 2333
pthread_setcanceltype() 2335
pthread_setconcurrency() 2337
pthread_setname_np() 2339
pthread_setschedparam() 2341
pthread_setspecific() 2343
pthread_sigmask() 2345
pthread_sleepon_broadcast() 2347
pthread_sleepon_lock() 2349
pthread_sleepon_signal() 2351
pthread_sleepon_timedwait() 2353
pthread_sleepon_unlock() 2357
pthread_sleepon_wait() 2359
pthread_spin_destroy() 2363
pthread_spin_init() 2365
pthread_spin_lock() 2367
pthread_spin_trylock() 2369
pthread_spin_unlock() 2371
pthread_testcancel() 2373
pthread_timedjoin() 2374
pulse 2377
pulse_attach() 2379
pulse_detach() 2383
putc() 2386
putc_unlocked() 2388
putchar() 2390
putchar_unlocked() 2392
putenv() 2394
puts() 2397
putspent() 2399
putstring() 2402
putw() 2405
putwc() 2407
putwchar() 2409
pwrite(), pwrite64() 2411
qnx_crypt() 2414
qsort() 2417
Raccept() 2421
raise() 2423
rand() 2426
rand_r() 2428
random() 2430
Rbind() 2433
rcmd() 2435
Rconnect() 2438
rdchk() 2440
re_comp() 2442
re_exec() 2444
read() 2446
read_main_config_file() 2451
readblock() 2455
readcond() 2458
readdir() 2464
readdir_r() 2468
readlink() 2471
readv() 2474
realloc() 2478
realpath() 2481
recv() 2483
recvfrom() 2486
recvmsg() 2490
regcomp() 2494
regerror() 2499
regexec() 2501
regfree() 2505
remainder(), remainderf() 2507
remove() 2509
rename() 2512
res_init() 2515
res_mkquery() 2518
res_query() 2521
res_querydomain() 2524
res_search() 2527
res_send() 2530
resmgr_attach() 2533
resmgr_block() 2542
resmgr_connect_funcs_t 2545
resmgr_context_alloc() 2547
resmgr_context_free() 2550
resmgr_context_t 2552
resmgr_detach() 2554
resmgr_devino() 2558
_resmgr_handle_grow() 2561
resmgr_handle_tune() 2563
resmgr_handler() 2566
_resmgr_io_func() 2569
resmgr_io_funcs_t 2571
resmgr_iофunсs() 2576
resmgr_msgread() 2578
resmgr_msgreadv() 2580
resmgr_msgwrite() 2582
resmgr_msgwritev() 2584
_RESMGR_NPARTS() 2586
_resmgr_ocб() 2588
resmgr_open_bind() 2590
resmgr_pathname() 2593
_RESMGR_PTR() 2596
_RESMGR_STATUS() 2598
resmgr_unbind() 2600
rewind() 2600
rewinddir() 2605
Rgetsockname() 2608
rindex() 2610
rint(), printf() 2612
Rlisten() 2615
rmdir() 2617
ROUTE 2620
Rcmd() 2627
rresvport() 2629
Rselect() 2631
rsrcremgr_attach() 2633
rsrcremgr_create() 2640
rsrcremgr_destroy() 2644
rsrcremgr_detach() 2646
rsrcremgr_devno_attach() 2648
rsrcremgr_devno_detach() 2652
rsrcremgr_query() 2654
ruserok() 2657

C Library — S 2659
sbrk() 2662
scalb(), scalbf() 2665
scalbn(), scalbnf() 2668
_scalloc() 2671
scandir() 2673
scanf() 2676
sched_getparam() 2685
sched_get_priority_adjust() 2688
sched_get_priority_max() 2690
sched_get_priority_min() 2692
sched_setscheduler() 2694
sched_param 2696
sched_rr_get_interval() 2702
sched_setscheduler() 2704
sched_setscheduler() 2707
sched_yield() 2710
SchedCtl(), SchedCtl_r() 2713
SchedGet(), SchedGet_r() 2747
SchedInfo(), SchedInfo_r() 2750
SchedSet(), SchedSet_r() 2753
SchedYield(), SchedYield_r() 2757
SCTP 2759
sctp_bindx() 2761
sctp_connectx() 2764
sctp_freeladdr() 2766
sctp_freepaddr() 2767
sctp_getladdr() 2769
sctp_getpaddr() 2771
sctp_peeloff() 2773
sctp_recvmmsg() 2775
sctp_sendmmsg() 2778
searchenv() 2782
seed48() 2785
seekdir() 2787
select() 2789
select_attach() 2795
select_detach() 2799
select_query() 2802
sem_close() 2805
sem_destroy() 2807
sem_getvalue() 2809
sem_init() 2811
sem_open() 2814
sem_post() 2819
sem_timedwait() 2821
sem_trywait() 2824
sem.unlink() 2826
sem.wait() 2828
send() 2830
sendmsg() 2833
sendto() 2836
servent 2839
setbuf() 2840
setbuffer() 2842
setdomainname() 2845
setegid() 2847
setenv() 2850
seteuid() 2853
setgid() 2856
setgrent() 2859
setgroups() 2861
sethostent() 2863
sethostname() 2865
SETOV() 2867
setitimer() 2869
setjmp() 2872
setkey() 2875
setlinebuf() 2877
setlocale() 2879
setlogmask() 2882
setnetent() 2884
setpgid() 2886
setpgrp() 2889
setprio() 2891
setprotoent() 2893
setpwent() 2895
setregid() 2896
setreuid() 2899
setrlimit(), setrlimit64()  2902
setservent()  2909
setsid()  2911
setsockopt()  2913
setspent()  2916
setstate()  2917
settimeofday()  2919
setuid()  2921
setutent()  2924
setvbuf()  2926
_sfree()  2929
shm_ctl()  2931
shm_ctl_special()  2940
shm_open()  2943
shm_unlink()  2950
shutdown()  2952
sigaction()  2954
sigaddset()  2960
sigblock()  2962
sigdelset()  2964
sigemptyset()  2966
sigevent  2968
sigfillset()  2975
sigismember()  2977
siglongjmp()  2979
sigmask()  2981
signal()  2983
SignalAction(), SignalAction_r()  2987
SignalKill(), SignalKill_r()  2995
SignalProcmask(), SignalProcmask_r()  3001
SignalSuspend(), SignalSuspend_r()  3006
SignalWaitinfo(), SignalWaitinfo_r()  3009
significand(), significandf() 3012
sigpause() 3015
sigpending() 3017
sigprocmask() 3019
sigqueue() 3022
sigsetjmp() 3025
sigsetmask() 3027
sigsuspend() 3029
sigtimedwait() 3031
sigunblock() 3034
sigwait() 3036
sigwaitinfo() 3038
sin(), sinf() 3040
sinh(), sinh() 3042
sleep() 3044
__sleepon_broadcast() 3046
__sleepon_destroy() 3048
__sleepon_init() 3050
__sleepon_lock() 3052
__sleepon_signal() 3054
__sleepon_unlock() 3056
__sleepon_wait() 3058
slogb() 3060
slogf() 3062
slogi() 3066
__smalloc() 3068
snmp_close() 3070
snmp_free_pdu() 3072
snmp_open() 3074
snmp_pdu 3076
snmp_pdu_create() 3080
snmp_read() 3082
snmp_select_info() 3084
snmp_send() 3087
snmp_session 3090
snmp_timeout() 3094
snprintf() 3096
sockatmark() 3099
socket() 3101
socketpair() 3105
SOCKSinit() 3108
sopen() 3110
sopenfd() 3115
spawn() 3118
spawnl() 3129
spawnle() 3134
spawnlp() 3139
spawnlpe() 3144
spawnp() 3149
spawnv() 3156
spawnve() 3161
spawnvp() 3166
spawnvpe() 3170
sprintf() 3175
sqrt(), sqrtf() 3178
srand() 3180
srand48() 3182
srandom() 3184
_srealloc() 3186
sscanf() 3189
stat(), stat64() 3191
statvfs(), statvfs64() 3199
stderr 3203
stdin 3204
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdout</td>
<td>3205</td>
</tr>
<tr>
<td>straddstr()</td>
<td>3206</td>
</tr>
<tr>
<td>strcasecmp()</td>
<td>3208</td>
</tr>
<tr>
<td>strcat()</td>
<td>3211</td>
</tr>
<tr>
<td>strchr()</td>
<td>3213</td>
</tr>
<tr>
<td>strcmp()</td>
<td>3215</td>
</tr>
<tr>
<td>strcmpi()</td>
<td>3217</td>
</tr>
<tr>
<td>strcoll()</td>
<td>3219</td>
</tr>
<tr>
<td>strcpy()</td>
<td>3221</td>
</tr>
<tr>
<td>strcspn()</td>
<td>3223</td>
</tr>
<tr>
<td>strdup()</td>
<td>3225</td>
</tr>
<tr>
<td>strerror()</td>
<td>3227</td>
</tr>
<tr>
<td>strftime()</td>
<td>3229</td>
</tr>
<tr>
<td>stricmp()</td>
<td>3235</td>
</tr>
<tr>
<td>strlen()</td>
<td>3237</td>
</tr>
<tr>
<td>strlwr()</td>
<td>3239</td>
</tr>
<tr>
<td>strncasecmp()</td>
<td>3241</td>
</tr>
<tr>
<td>strncat()</td>
<td>3244</td>
</tr>
<tr>
<td>strncmp()</td>
<td>3246</td>
</tr>
<tr>
<td>strncpy()</td>
<td>3248</td>
</tr>
<tr>
<td>strnicmp()</td>
<td>3251</td>
</tr>
<tr>
<td>strlen()</td>
<td>3253</td>
</tr>
<tr>
<td>strpbrk()</td>
<td>3255</td>
</tr>
<tr>
<td>strrchr()</td>
<td>3257</td>
</tr>
<tr>
<td>strrev()</td>
<td>3259</td>
</tr>
<tr>
<td>strstr()</td>
<td>3261</td>
</tr>
<tr>
<td>strdup()</td>
<td>3263</td>
</tr>
<tr>
<td>strslen()</td>
<td>3265</td>
</tr>
<tr>
<td>strspn()</td>
<td>3267</td>
</tr>
<tr>
<td>strr()</td>
<td>3269</td>
</tr>
<tr>
<td>strtod()</td>
<td>3271</td>
</tr>
<tr>
<td>strtoimax(), strtoimax()</td>
<td>3273</td>
</tr>
</tbody>
</table>
tanh(), tanhf()  3364
tcdrain()  3366
tcdropline()  3368
tcflow()  3371
tcflush()  3374
tcgetattr()  3377
tcgetpgrp()  3379
tcgetsid()  3381
tcgetsize()  3383
tcinject()  3385
tciscurs()  3388
TCP  3390
tcsendbreak()  3393
tcsetattr()  3396
tcsetpgrp()  3399
tcsetsid()  3402

tcsetsize()  3404
tell(), tell64()  3406
telldir()  3409
tempnam()  3411
termios  3413

thread_pool_control()  3417
thread_pool_create()  3420
thread_pool_destroy()  3427
thread_pool_limits()  3430

thread_pool_start()  3433
ThreadCancel(), ThreadCancel_r()  3436
ThreadCreate(), ThreadCreate_r()  3440
ThreadCtl(), ThreadCtl_r()  3447
ThreadDestroy(), ThreadDestroy_r()  3455
ThreadDetach(), ThreadDetach_r()  3458
ThreadJoin(), ThreadJoin_r()  3460
time() 3463
timer_create() 3465
timer_delete() 3469
timer_getexpstatus() 3471
timer_getoverrun() 3473
timer_gettime() 3475
timer_settime() 3477
timer_timeout(), timer_timeout_r() 3480
TimerAlarm(), TimerAlarm_r() 3488
TimerCreate(), TimerCreate_r() 3491
TimerDestroy(), TimerDestroy_r() 3495
TimerInfo(), TimerInfo_r() 3497
TimerSettime(), TimerSettime_r() 3501
TimerTimeout(), TimerTimeout_r() 3505
times() 3513
timespec 3516
timespec2nsec() 3517
timezone 3519
tm 3520
tmpfile(), tmpfile64() 3522
tmpnam() 3525
tolower() 3528
toupper() 3530
towctrans() 3532
towlower() 3534
towupper() 3536
tracebuf 3538
TraceEvent() 3540
traceheader 3556
traceparser() 3558
traceparser_cs() 3560
traceparser_cs_range() 3562
traceparser_debug() 3564
traceparser_destroy() 3566
traceparser_get_info() 3567
traceparser_init() 3571
truncate() 3573
ttyname() 3576
ttynamex() 3578
tzname 3580
tzset() 3581
ualarm() 3584
UDP 3587
ultoa(), ulloa() 3589
umask() 3592
umount() 3595
UNALIGNED_PUT16() 3597
UNALIGNED_PUT32() 3599
UNALIGNED_PUT64() 3601
UNALIGNED_RET16() 3603
UNALIGNED_RET32() 3605
UNALIGNED_RET64() 3607
uname() 3609
ungetc() 3612
ungetwc() 3614
UNIX 3616
unlink() 3619
unsetenv() 3622
usleep() 3624
utime() 3626
utimes() 3629
utmp 3632
utmpxname() 3634
utoa() 3636
va_arg() 3639
va_copy() 3645
va_end() 3647
va_start() 3649
valloc() 3651
verr(), verrx() 3653
vfork() 3655
vfprintf() 3657
vfscanf() 3660
vfwprintf() 3663
vfwsprintf() 3665
vprintf() 3667
vsprintf() 3669
vslogf() 3672
vsnprintf() 3674
vsprintf() 3677
vsscanf() 3680
vswprintf() 3683
vswscanf() 3685
vsyslog() 3687
vwarn(), vwarnx() 3689
vwprintf() 3691
vwscanf() 3693
wait() 3695
wait3() 3698
wait4() 3701
waitid() 3705
waitpid() 3708
warn(), warnx() 3711
wcrtomb() 3713
wcschr() 3715
weschr() 3717
wcscmp() 3719
wcscoll() 3721
wescpy() 3723
wescspn() 3725
wesftime() 3727
weslen() 3729
wesnecat() 3731
wesncmp() 3733
wesncpy() 3735
wespbrk() 3737
wesrchr() 3739
wesrtombs() 3741
wesspn() 3743
wesstr() 3745
wcestol(), wcestof(), wcestold() 3747
wcestoimax(), wcestoumax() 3750
wcestok() 3752
wcestol(), wcestoll() 3754
wcestombs() 3757
wcestoull() 3760
wescxfm() 3763
wctob() 3765
wctomb() 3767
wctrans() 3770
wctype() 3772
wmemchr() 3774
wmemcmp() 3776
wmemcpy() 3778
wmemmove() 3780
wmemset() 3782
wordexp() 3784
wordfree() 3786
wprintf() 3787
write() 3789
writeblock() 3794
writev() 3797
wscanf() 3800
y0(), y0f() 3802
y1(), y1f() 3804
yn(), ynf() 3806

A  SOCKS — A Basic Firewall 3809

About SOCKS 3811
How to SOCKSify a client 3811
What SOCKS expects 3812

B  Summary of Safety Information 3815

Cancellation points 3817
Interrupt handlers 3822
Signal handlers 3825
Multithreaded programs 3839

C  What’s New in this Reference? 3843

What’s new in QNX Momentics 6.3.2? 3845
    Errata 3845
What’s new in the QNX Neutrino Core OS 6.3.2? 3845
    New Content 3845
    Changed Content 3846
What’s new in QNX Neutrino 6.3.0 Service Pack 2? 3847
    New Content 3848
    Changed Content 3848
    Errata 3848
What’s new in QNX Neutrino 6.3.0 Service Pack 1? 3849
    New content 3849
    Changed content 3850
Errata 3852
What’s new in QNX Neutrino 6.3.0? 3852  
New content 3852
What’s new in QNX Neutrino 6.2.1? 3854  
New content 3854
Changed content 3855
Errata 3855
What’s new in QNX Neutrino 6.2? 3857  
New Content 3857
Deprecated Content 3860
Errata 3860
What’s new in the QNX Neutrino 6.1.0 docs? 3861  
New content 3861
Deprecated content 3862

D  Third-Party Copyright Notices 3863
 BSD stack 3865
 BSD stack and various utilities 3866
 MINIX operating system 3873
 ncurses library 3874
 Regular-expression handling 3875
 Remote Procedure Call (RPC) 3875
 SNMPv2 3876
 SOCKS 3877

Glossary 3879

Index 3903
List of Figures

A hierarchy of processes. 1763
A deadlock when sending messages improperly among processes. 1764

MsgSendv(), client to process manager. 1776
MsgReplyv(), process manager to client. 1776
MsgSendv(), client to filesystem manager 1777
Components of a fully qualified pathname. 1919
Specifying a guardian for child processes. 2115
Conditions that satisfy an input request. 2459

Most of the spawn*() functions do a lot of work before a message is sent to procnto. 3120
About This Reference
The *Library Reference* describes the C functions, data types, and protocols that are included as part of the QNX Neutrino RTOS.

<table>
<thead>
<tr>
<th>For information about:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The entries arranged into categories</td>
<td>Summary of Functions</td>
</tr>
<tr>
<td>An overview of what you’ll find in each entry</td>
<td>“What’s in a function description?” in the Summary of Functions chapter</td>
</tr>
<tr>
<td>Descriptions of the manifests that you can use for compile-time changes or inspection</td>
<td>Manifests</td>
</tr>
<tr>
<td>The SOCKS package</td>
<td>SOCKS — A Basic Firewall</td>
</tr>
<tr>
<td>A summary of functions that are cancellation points, that you can safely call from an interrupt handler, that you can safely call from a signal handler, and that you can’t safely call from a multithreaded program.</td>
<td>Summary of Safety Information</td>
</tr>
<tr>
<td>Changes to this reference</td>
<td>What’s New in this Reference?</td>
</tr>
<tr>
<td>Copyright information</td>
<td>Third-Party Copyright Notices</td>
</tr>
<tr>
<td>Terms used in QNX documentation</td>
<td>Glossary</td>
</tr>
</tbody>
</table>

**Typographical conventions**

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:
## Typographical conventions

<table>
<thead>
<tr>
<th>Reference</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code examples</td>
<td><code>if( stream == NULL )</code></td>
</tr>
<tr>
<td>Command options</td>
<td><code>-lR</code></td>
</tr>
<tr>
<td>Commands</td>
<td><code>make</code></td>
</tr>
<tr>
<td>Environment variables</td>
<td><code>PATH</code></td>
</tr>
<tr>
<td>File and pathnames</td>
<td><code>/dev/null</code></td>
</tr>
<tr>
<td>Function names</td>
<td><code>exit()</code></td>
</tr>
<tr>
<td>Keyboard chords</td>
<td><code>Ctrl-Alt-Delete</code></td>
</tr>
<tr>
<td>Keyboard input</td>
<td><code>something you type</code></td>
</tr>
<tr>
<td>Keyboard keys</td>
<td><code>Enter</code></td>
</tr>
<tr>
<td>Program output</td>
<td><code>login:</code></td>
</tr>
<tr>
<td>Programming constants</td>
<td><code>NULL</code></td>
</tr>
<tr>
<td>Programming data types</td>
<td><code>unsigned short</code></td>
</tr>
<tr>
<td>Programming literals</td>
<td><code>0xFF, &quot;message string&quot;</code></td>
</tr>
<tr>
<td>Variable names</td>
<td><code>stdin</code></td>
</tr>
<tr>
<td>User-interface components</td>
<td><code>Cancel</code></td>
</tr>
</tbody>
</table>

We use an arrow (→) in directions for accessing menu items, like this:

You’ll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:

---

Notes point out something important or useful.
CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

To obtain technical support for any QNX product, visit the Support + Services area on our website (www.qnx.com). You’ll find a wide range of support options, including community forums.
Summary of Functions
Summary of function categories

We’ve organized the functions in the C library into the following categories:

Asynchronous I/O functions
Asynchronous read, write, and other I/O operations.

Atomic functions
Thread-safe integer manipulation functions.

Character manipulation functions
Single-character functions for upper/lowercase conversions.

Conversion functions
Convert values from one representation to another (e.g. numeric values to strings).

Directory functions
Directory services (change, open, close, etc.).

Dispatch interface functions
Handle different event types, including messages, pulse codes, and signals.

File manipulation functions
File operations (change permissions, delete, rename, etc.)

IPC functions
Traditional InterProcess Communication functions.

Hardware functions
These functions work with PCI and other devices.

Math functions
Perform computations such as the common trigonometric calculations. These functions operate with floating-point values.
Memory allocation functions
    Allocate and deallocate memory.

Memory manipulation functions
    Manipulate blocks of memory.

Message-queue functions
    Nonblocking message-passing facilities.

Multibyte character functions
    ANSI C functions for processing multibyte and
    wide characters.

QNX Neutrino-specific IPC functions
    Native message-passing and related functions.

Operating system I/O functions
    POSIX functions for performing I/O at a lower
    level than the C Language stream I/O functions
    (e.g. \texttt{fopen()}, \texttt{fread()}, \texttt{fwrite()}, and \texttt{fclose()}).

PC Card functions
    Native PC Card functions.

Platform-specific functions
    Invoke Intel 80x86 and other processor-related
    functions directly from a program.

Process environment functions
    For process identification, user identification,
    process groups, system identification, system time
    and process time, environment variables, terminal
    identification, and configurable system variables.

Process manipulation functions
    For process creation, execution, and termination;
    signal handling; and timer operations.
Realtime timer functions
Rich set of “inexpensive” timer functions that are quick to create and manipulate.

Resource manager functions
These functions help you create resource managers.

Searching and sorting functions
Perform various search and sort operations (do a binary search on a sorted array, find one string inside another, etc.).

Shared-memory functions
Create and manipulate shared-memory regions.

Signal functions
Rich set of functions for handling and sending signals.

Stream I/O functions
The “standard” functions to read and write files. Data can be transmitted under format control or as characters, strings, or blocks of memory.

String manipulation functions
Manipulate a character string, i.e. an array of zero or more adjacent characters followed by a NUL character (\0) that marks the end of the string.

System database functions
Allow an application to access group and user database information.

System message log functions
This set of functions controls the system log.

TCP/IP functions
Handle TCP/IP network communications and the TCP/IP database files.
Summary of function categories

Terminal control functions
Set and control terminal attributes (baud rate, flow control, etc.).

Thread functions
Operate on threads and the objects used to synchronize threads.

Time functions
Obtain and manipulate times and dates.

Variable-length argument list functions
Process a variable number of arguments to a function.

Wide-character functions
Wide-character versions of functions from other function summary categories.

The following subsections describe these function categories in more detail. Each function is noted with a brief description of its purpose.

Asynchronous I/O functions
These functions perform asynchronous read, write, and other I/O operations.

Asynchronous I/O operations aren’t currently supported.

The following functions are defined:

- `aio_cancel()` Cancel an asynchronous I/O operation
- `aio_error()` Get the error status for an asynchronous I/O operation
- `aio_fsync()` Asynchronously synchronize a file
- `aio_read()` Asynchronously read from a file
аіо_реurt()  Get the return status for an asynchronous І/О operation
аіо_пеn( )  Wait for asynchronous І/О operations to complete
аіо_вгіт()  Asynchronously write to a file

Atomic functions

These functions manipulate an integer in a thread-safe way. On a multiprocessor system, even a simple:

```c
/*
   Assuming x is an unsigned variable shared between two
   or more threads or a thread and an interrupt handler.
*/
x ^= 0xdeadbeef;
```

may cause x to be in an undefined state if multiple threads running simultaneously on multiple processors execute this code at the same time.

Use the `atomic*( )` functions to ensure that your integer operations are carried out properly:

```c
atomic_toggle( &x, 0xdeadbeef );
```

atomic_add()  Safely add to a variable
atomic_add_value()  Safely add to a variable, returning the previous value
atomic_clr()  Safely clear a variable
atomic_clr_value()  Safely clear a variable, returning the previous value
atomic_set()  Safely set bits in a variable
atomic_set_value()  
Safely set bits in a variable, returning the previous value

atomic_sub()  
Safely subtract from a variable

atomic_sub_value()  
Safely subtract from a variable, returning the previous value

atomic_toggle()  
Safely toggle a variable

atomic_toggle_value()  
Safely toggle a variable, returning the previous value

Character manipulation functions

These functions operate on single characters of type char. The functions test characters in various ways and convert them between upper and lowercase. (Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.)

isalnum()  
Test a character to see if it’s alphanumeric

isalpha()  
Test to see if a character is a letter

isascii()  
Test for a character in the range 0 to 127

iscntrl()  
Test a character to see if it’s a control character

isdigit()  
Test for any decimal digit

isgraph()  
Test for any printable character except space

islower()  
Test for any lowercase letter

isprint()  
Test for any printable character, including space
### Summary of function categories

- **ispunct**(): Test for any punctuation character
- **isspace**(): Test for a whitespace character
- **isupper**(): Test for any uppercase letter
- **isxdigit**(): Test for any hexadecimal digit
- **tolower**(): Convert a character to lowercase
- **toupper**(): Convert a character to uppercase

### Conversion functions

These functions perform conversions between objects of various types and strings:

- **atof**(): Convert a string into a `double`
- **atoh**(): Convert a string containing a hexadecimal number into an unsigned number
- **atoi**(): Convert a string into an integer
- **atol**, **atoll**(): Convert a string into a long integer

#### ENDIAN

- **ENDIAN_BE16()**: Return a big-endian 16-bit value in native format
- **ENDIAN_BE32()**: Return a big-endian 32-bit value in native format
- **ENDIAN_BE64()**: Return a big-endian 64-bit value in native format
- **ENDIAN_LE16()**: Return a little-endian 16-bit value in native format
- **ENDIAN_LE32()**: Return a little-endian 32-bit value in native format
ENDIAN_LE64()  
Return a little-endian 64-bit value in native format

ENDIAN_RET16()  
Return an endian-swapped 16-bit value

ENDIAN_RET32()  
Return an endian-swapped 32-bit value

ENDIAN_RET64()  
Return an endian-swapped 64-bit value

ENDIAN_SWAP16()  
Endian-swap a 16-bit value in place

ENDIAN_SWAP32()  
Endian-swap a 32-bit value in place

ENDIAN_SWAP64()  
Endian-swap a 64-bit value in place

htonl()  
Convert a 32-bit value from host-byte order to network-byte order

htons()  
Convert a 16-bit value from host-byte order to network-byte order

itoa()  
Convert an integer into a string, using a given base

ltoa(), lltoa()  
Convert a long integer value into a string, using a given base

nsec2timespec()  
Convert nanoseconds to a timespec structure

ntohl()  
Convert network-byte order value

ntohs()  
Convert network-byte order value

strtod()  
Convert a string into a double
Summary of function categories

strtoimax(), strtoumax()
Convert a string into an integer

strtol(), strtold()
Convert a string into a long integer

strtooul(), strtooull()
Convert a string into an unsigned long integer

timespec
timespec2nsec()
Convert a timespec structure to nanoseconds

ultoa(), ulltoa() Convert an unsigned long integer into a string, using a given base

UNALIGNED_PUT16()
Write a misaligned 16-bit value safely

UNALIGNED_PUT32()
Write a misaligned 32-bit value safely

UNALIGNED_PUT64()
Write a misaligned 64-bit value safely

UNALIGNED_RET16()
Access a misaligned 16-bit value safely

UNALIGNED_RET32()
Access a misaligned 32-bit value safely

UNALIGNED_RET64()
Access a misaligned 64-bit value safely

utoa()
Convert an unsigned integer into a string, using a given base

wordexp() Perform word expansions
Summary of function categories

wordfree() Free a word expansion buffer

See also the following functions, which convert the cases of characters and strings:

- strlwr()
- strupr()
- tolower()
- toupper()

Directory functions

These functions pertain to directory manipulation:

- alphasort() Compare two directory entries
- chdir() Change the current working directory
- chroot() Change the root directory
- closedir() Close a directory
- directl() Control an open directory
- dirname() Report the parent directory name of a file pathname
- getcwd() Get the name of the current working directory
- getwd() Get current working directory pathname
- glob() Find paths matching a pattern
- globfree() Free storage allocated by a call to glob()
- mkdir() Create a subdirectory
- mount() Mount a filesystem

mount_parse_generic_args() Strip off common mount arguments
opendir()  Open a directory file

pathfind(), pathfind_r()  
Search for a file in a list of directories

readdir()  Get information about the next matching filename

readdir_r()  Get information about the next matching filename

realpath()  Resolve a pathname

rewinddir()  Reset the position of a directory stream to the start of the directory

rmdir()  Delete an empty directory

scandir()  Scan a directory

seekdir()  Set the position for the next read of the directory stream

telldir()  Get the location associated with the directory stream

umount()  Unmount a filesystem

## Dispatch interface functions

These functions make up the dispatch interface where you can handle different event types including messages, pulse codes, and signals. The functions cover dispatch contexts, attaching events, attaching pathnames and file descriptors to dispatch contexts, thread pools, etc. For an overview of these functions, see “Components of a resource manager” in the Writing a Resource Manager chapter of the QNX Neutrino Programmer’s Guide.

dispatch_block()  
Block while waiting for an event

dispatch_context_alloc()  
Return a dispatch context
Summary of function categories

(dispatch_context_free())
Free a dispatch context

(dispatch_create())
Allocate a dispatch handle

(dispatch_destroy())
Destroy a dispatch handle

(dispatch_handler())
Handle events received by dispatch_block()

(dispatch_timeout())
Set a timeout

(dispatch_unblock())
Unblock all of the threads that are blocked on a dispatch handle

(message_attach())
Attach a message range

(message_connect())
Create a connection to a channel

(message_detach())
Detach a message range

(name_attach())
Register a name in the namespace and create a channel

(name_detach())
Remove a name from the namespace and destroy the channel

_pulse Structure that describes a pulse

(pulse_attach())
Attach a handler function to a pulse code
Summary of function categories

*pulse_detach()*

Detach a handler function from a pulse code

*resmgr_attach()*

Attach a path to a pathname space

*resmgr_block()*

Block while waiting for a message

*resmgr_connect_funcs_t*

Table of POSIX-level connect functions

*resmgr_context_alloc()*

Allocate a resource-manager context

*resmgr_context_free()*

Free a resource-manager context

*resmgr_context_t*

Context information that’s passed between resource-manager functions

*resmgr_detach()*

Remove a pathname from the pathname space

*resmgr_devino()*

Get the device and inode number

*_resmgr_handle_grow()*

Expand the capacity of the device manager database

*resmgr_handle_tune()*

Tune aspects of client fd-to-OCB mapping

*resmgr_handler()*

Handle resource manager messages

*_resmgr_io_func()*

Retrieve an I/O function from an I/O function table
**resmgr_io_funcs_t**

Table of POSIX-level I/O functions

*resmgr_iофuncs()*

Extract the I/O function pointers associated with client connections

*resmgr_msgread()*

Read a message from a client

*resmgr_msgreadv()*

Read a message from a client

*resmgr_msgwrite()*

Write a message to a client

*resmgr_msgwritev()*

Write a message to a client

*_RESMGR_NPARTS()*

Get a given number of parts from the ctp->iov structure

*_resmgr_ocb()*

Retrieve an Open Control Block

*resmgr_open_bind()*

Associate an OCB with an open request

*resmgr_pathname()*

Return the pathname associated with an ID

*_RESMGR_PTR()*

Get one part from the ctp->iov structure and fill in its fields

*_RESMGR_STATUS()*

Set the status member of a resource-manager context
resmgr_unbind()
   Remove an OCB

select_attach()
   Attach a file descriptor to a dispatch handle

select_detach()
   Detach a file descriptor from a dispatch handle

select_query()
   Decode the last select event

thread_pool_create()
   Create a thread pool handle

thread_pool_control()
   Control the thread pool behavior

thread_pool_destroy()
   Free the memory allocated to a thread pool

thread_pool_limits()
   Wrapper function for thread_pool_control()

thread_pool_start()
   Start a thread pool

File manipulation functions

These functions operate directly with files. The following functions are defined:

access() Check to see if a file or directory can be accessed
chmod() Change the permissions for a file
chown() Change the user ID and group ID of a file
eaccess() Check to see if a file or directory can be accessed (extended version)
### Summary of Functions

- **glob()** Find paths matching a pattern
- **globfree()** Free storage allocated by a call to `glob()`
- **fchmod()** Change the permissions for a file
- **fchown()** Change the user ID and group ID of a file
- **fpathconf()** Return the value of a configurable limit associated with a file or directory
- **ftruncate(), ftruncate64()** Truncate a file
- **futime()** Record the modification time for a file
- **lchown()** Change the user ID and group ID of a file or symbolic link
- **lstat(), lstat64()** Get information about a file or directory
- **ltrunc()** Truncate a file at a given position
- **mkfifo()** Create a FIFO special file
- **mkstemp()** Make a unique temporary filename, and open the file
- **mktemp()** Make a unique temporary filename
- **nftw(), nftw64()** Walk a file tree
- **pathconf()** Return the value of a configurable limit
- **pclose()** Close a pipe
- **pwrite(), pwrite64()** Write into a file without changing the file pointer
- **remove()** Remove a link to a file
rename() Rename a file

stat(), stat64()
Get information about a file or directory, given a path

statvfs(), statvfs64()
Get filesystem information, given a path

sync() Synchronize filesystem updates

tempnam() Create a name for a temporary file

truncate() Truncate a file to a specified length

tmpnam() Generate a unique string for use as a filename

unlink() Remove a link to a file

utime() Record the modification time for a file or directory

utimes() Set a file’s access and modification times

**IPC functions**

These functions deal with InterProcess Communications.

flock() Apply or remove an advisory lock on an open file

lockf() Record locking on files

mlock() Lock a buffer in physical memory

mlockall() Lock a process’s address space

mmap(), mmap64() Map a memory region into a process address space

mprotect() Change memory protection

msync() Synchronize memory with physical storage

munlock() Unlock a buffer

September 10, 2007
Summary of function categories

munlockall()  Unlock a process’s address space
munmap()  Unmap previously mapped addresses

pthread_barrier_destroy()  
Destroy a barrier object

pthread_barrier_init()  
Initialize a barrier object

pthread_barrier_wait()  
Synchronize at a barrier

pthread_barrierattr_destroy()  
Destroy a barrier attributes object

pthread_barrierattr_getpshared()  
Get the process-shared attribute of a barrier attributes object

pthread_barrierattr_init()  
Initialize a barrier attributes object

pthread_barrierattr_setpshared()  
Set the process-shared attribute of a barrier attributes object

pthread_cond_broadcast()  
Unblock threads waiting on a condition

pthread_cond_destroy()  
Destroy a condition variable

pthread_cond_init()  
Initialize a condition variable

pthread_cond_signal()  
Unblock the thread waiting on a condition variable
Summary of function categories

`pthread_cond_timedwait()`  
Wait on a condition variable, with a time limit

`pthread_cond_wait()`  
Wait on a condition variable

`pthread_condattr_destroy()`  
Destroy a condition variable attribute object

`pthread_condattr_getclock()`  
Get the clock attribute from a condition-variable attribute object

`pthread_condattr_getpshared()`  
Get the process-shared attribute from a condition variable attribute object

`pthread_condattr_init()`  
Initialize a condition variable attribute object

`pthread_condattr_setclock()`  
Set the clock attribute in a condition-variable attribute object

`pthread_condattr_setpshared()`  
Set the process-shared attribute in a condition-variable attribute object

`pthread_mutex_destroy()`  
Destroy a mutex

`pthread_mutex_getprioceiling()`  
Get a mutex’s priority ceiling

`pthread_mutex_init()`  
Initialize a mutex

`pthread_mutex_lock()`  
Lock a mutex
Summary of function categories

pthread_mutex_setprioceiling()  
    Set a mutex’s priority ceiling

pthread_mutex_timedlock()  
    Lock a mutex

pthread_mutex_trylock()  
    Attempt to lock a mutex

pthread_mutex_unlock()  
    Unlock a mutex

pthread_mutexattr_destroy()  
    Destroy a mutex attribute object

pthread_mutexattr_getprioceiling()  
    Get the priority ceiling of a mutex attribute object

pthread_mutexattr_getprotocol()  
    Get a mutex’s scheduling protocol

pthread_mutexattr_getpshared()  
    Get the process-shared attribute from a mutex attribute object

pthread_mutexattr_getrecursive()  
    Get the recursive attribute from a mutex attribute object

pthread_mutexattr_gettype()  
    Get a mutex type

pthread_mutexattr_init()  
    Initialize the mutex attribute object

pthread_mutexattr_setprioceiling()  
    Set the priority ceiling of a mutex attribute object
Summary of function categories

*pthread_mutexattr_setprotocol()*

Set a mutex’s scheduling protocol

*pthread_mutexattr_setpshared()*

Set the process-shared attribute in mutex attribute object

*pthread_mutexattr_setrecursive()*

Set the recursive attribute in mutex attribute object

*pthread_mutexattr_settype()*

Set a mutex type

*pthread_once() Dynamic package initialization*

*pthread_rwlock_destroy()*

Destroy a read/write lock

*pthread_rwlock_init()*

Initialize a read/write lock

*pthread_rwlock_rdlock()*

Acquire a shared read lock on a read/write lock

*pthread_rwlock_timedrdlock()*

Lock a read-write lock for writing

*pthread_rwlock_timedwrlock()*

Attempt to acquire an exclusive write lock on a read/write lock

*pthread_rwlock_tryrdlock()*

Attempt to acquire a shared read lock on a read/write lock

*pthread_rwlock_trywrlock()*

Attempt to acquire an exclusive write lock on a read/write lock
Summary of function categories

pthread_rwlock_unlock()
Unlock a read/write lock

pthread_rwlock_wrlock()
Acquire an exclusive write lock on a read/write lock

pthread_rwlockattr_destroy()
Destroy a read-write lock attribute object

pthread_rwlockattr_getpshared()
Get the process-shared attribute of a read-write lock attribute object

pthread_rwlockattr_init()
Create a read-write lock attribute object

pthread_rwlockattr_setpshared()
Set the process-shared attribute of a read-write lock attribute object

pthread_spin_destroy()
Destroy a thread spinlock

pthread_spin_init()
Initialize a thread spinlock

pthread_spin_lock()
Lock a thread spinlock

pthread_spin_trylock()
Try locking a thread spinlock

pthread_spin_unlock()
Unlock a thread spinlock

readcond()  Read data from a terminal device

sem_close()  Close a named semaphore
Summary of function categories

- `sem_destroy()`  Destroy a semaphore
- `sem_getvalue()` Get the value of a semaphore (named or unnamed)
- `sem_init()`  Initialize a semaphore
- `sem_open()`  Create or access a named semaphore
- `sem_post()`  Increment a semaphore
- `sem_timedwait()`  Wait on a semaphore, with a timeout
- `sem_trywait()`  Wait on a semaphore, but don’t block
- `sem_unlink()`  Destroy a named semaphore
- `sem_wait()`  Wait on a semaphore
- `sync()`  Synchronize filesystem updates

Hardware functions

These functions work with PCI and other devices for operations such as determining whether or not a PCI BIOS is present, attaching a driver to a PCI device, and so on.

The following functions are defined:

- `pci_attach()`  Connect to the PCI server
- `pci_attach_device()`  Attach a driver to a PCI device
- `pci_detach()`  Disconnect from the PCI server
- `pci_detach_device()`  Detach a driver from a PCI device
- `pci_find_class()`  Find devices that have a specific Class Code
Summary of function categories

`pci_find_device()`  
Find the PCI device with a given device ID and vendor ID

`pci_irq_routing_options()`  
Retrieve PCI IRQ routing information

`pci_map_irq()`  
Map an interrupt pin to an IRQ

`pci_present()`  
Determine whether or not PCI BIOS is present

`pci_read_config()`  
Read from the configuration space of a PCI device

`pci_read_config8()`  
Read a byte from the configuration space of a device

`pci_read_config16()`  
Read 16-bit values from the configuration space of a device

`pci_read_config32()`  
Read 32-bit values from the configuration space of a device

`pci_rescan_bus()`  
Rescan the PCI bus for added or removed devices

`pci_write_config()`  
Write to the configuration space of a PCI device

`pci_write_config8()`  
Write bytes to the configuration space of a PCI device

`pci_write_config16()`  
Write 16-bit values to the configuration space of a device
pci_write_config32()  Write 32-bit values to the configuration space of a device

hwi_find_item()  Find an item in the hwi_item structure

hwi_find_tag()  Find a tag in the hwi_item structure

hwi_off2tag()  Return a pointer to the start of a tag in the hwinfo area of the system page

hwi_tag2off()  Return the offset from the start of the hwinfo area of the system page

Math functions

The math functions are arranged in the following categories:

- Absolute values
- Bessel functions
- Divisions, remainders, and modular arithmetic
- Floating-point settings
- Gamma functions
- Logarithms and exponentials
- Miscellaneous
- Pseudo-random numbers
- Roots and powers
- Rounding
- Trigonometric and hyperbolic functions
Summary of function categories © 2007, QNX Software Systems GmbH & Co. KG.

Absolute values

\( \text{abs()} \)  
Return the absolute value of an integer

\( \text{cabs()}, \text{cabsf()} \)  
Compute the absolute value of a complex number

\( \text{fabs()}, \text{fabsf()} \)  
Compute the absolute value of a double number

\( \text{labs()} \)  
Calculate the absolute value of a long integer

Bessel functions

\( \text{j0()}, \text{j0f()} \)  
Compute a Bessel function of the first kind

\( \text{j1()}, \text{j1f()} \)  
Compute a Bessel function of the first kind

\( \text{jn()}, \text{jnf()} \)  
Compute a Bessel function of the first kind

\( \text{y0()}, \text{y0f()} \)  
Compute a Bessel function of the second kind

\( \text{y1()}, \text{y1f()} \)  
Compute a Bessel function of the second kind

\( \text{yn()}, \text{ynf()} \)  
Compute a Bessel function of the second kind

Division, remainders, and modular arithmetic

\( \text{div()} \)  
Calculate the quotient and remainder of a division operation

\( \text{drem()}, \text{dremf()} \)  
Compute the remainder of two numbers

\( \text{fmod()}, \text{fmodf()} \)  
Compute a residue, using floating-point modular arithmetic

\( \text{ldiv()} \)  
Perform division on long integers

\( \text{modf()}, \text{modff()} \)  
Break a number into integral and fractional parts

\( \text{remainder()}, \text{remainderf()} \)  
Compute the floating point remainder
Floating-point settings

These functions set or get attributes of floating-point operations:

- \texttt{fp\_exception\_mask()} Get or set the current exception mask
- \texttt{fp\_exception\_value()} Get the value of the current exception registers
- \texttt{fp\_precision()} Set or get the current precision
- \texttt{fp\_rounding()} Set or get the current rounding

Gamma functions

- \texttt{gamma()}, \texttt{gamma\_r()}, \texttt{gammaf()}, \texttt{gammaf\_r()} Log gamma function
- \texttt{lgamma()}, \texttt{lgamma\_r()}, \texttt{lgammaf()}, \texttt{lgammaf\_r()} Log gamma function

Logarithms and exponentials

The following routines calculate logarithms and exponentials:

- \texttt{exp()}, \texttt{expf()} Compute the exponential function of a number
- \texttt{expm1()}, \texttt{expm1f()} Compute the exponential of a number, then subtract 1
- \texttt{frexp()}, \texttt{frexpf()} Break a floating-point number into a normalized fraction and an integral power of 2
- \texttt{ilogb()}, \texttt{ilogbf()} Compute the integral part of a logarithm
- \texttt{ldexp()}, \texttt{ldexpf()} Multiply a floating-point number by an integral power of 2
Summary of function categories

\[ log() \], \[ logf() \] 
Compute the natural logarithm of a number

\[ log10() \], \[ log10f() \] 
Compute the logarithm (base 10) of a number

\[ log1p() \], \[ log1pf() \] 
Compute \( \log(1+x) \)

\[ logb() \], \[ logbf() \] 
Compute the radix-independent exponent

\[ scalb() \], \[ scalbf() \] 
Load the exponent of a radix-independent floating-point number

\[ scalbn() \], \[ scalbnf() \] 
Compute the exponent of a radix-independent floating-point number

\[ significand() \], \[ significandf() \] 
Compute the “significant bits” of a floating-point number

Miscellaneous

\[ copysign() \], \[ copysignf() \] 
Copy the sign bit from one number to another

\[ erf() \], \[ erff() \] 
Compute the error function of a number

\[ erfc() \], \[ erfcf() \] 
Complementary error function

\[ finite() \], \[ finitef() \] 
Determine if a number is finite

\[ hypot() \], \[ hypotf() \] 
Calculate the length of the hypotenuse for a right-angled triangle

\[ isinf() \], \[ isnffi() \] 
Test for infinity

\[ isnan() \], \[ isnanf() \] 
Test for not-a-number (NAN)

\[ max() \] 
Return the greater of two numbers

\[ min() \] 
Return the lesser of two numbers

\[ nextafter() \], \[ nextafterf() \] 
Compute the next representable double-precision floating-point number
Pseudo-random numbers

The math library includes several sets of functions that you can use to generate pseudo-random numbers.

The simplest family consists of:

- **rand()**: Compute a sequence of pseudo-random integers
- **rand_r()**: Compute a sequence of pseudo-random integers in a thread-safe manner
- **srand()**: Start a new sequence of pseudo-random integers for **rand()**

This set of functions uses a nonlinear additive feedback random-number generator, using a state array:

- **initstate()**: Initialize a pseudo-random number generator
- **random()**: Generate a pseudo-random number from the default state
- **setstate()**: Reset the state of a pseudo-random number generator
- **srandom()**: Set the seed for a pseudo-random number generator

This set of functions uses 48-bit arithmetic to produce pseudo-random numbers of various types:

- **drand48()**: Generate a pseudo-random **double**
- **erand48()**: Generate a pseudo-random **double** in a thread-safe manner
- **jrand48()**: Generate a pseudo-random signed **long** integer in a thread-safe manner
- **lcong48()**: Initialize a sequence of pseudo-random numbers
Summary of function categories

\textit{lrand48()}\quad Generate a pseudo-random nonnegative \texttt{long} integer

\textit{mrand48()}\quad Generate a pseudo-random signed \texttt{long} integer

\textit{nrand48()}\quad Generate a pseudo-random nonnegative \texttt{long} integer in a thread-safe manner

\textit{seed48()}\quad Initialize a sequence of pseudo-random numbers

\textit{srand48()}\quad Initialize a sequence of pseudo-random numbers

\textbf{Roots and powers}

\textit{cbrt()}, \textit{cbrtf()}\quad Compute the cube root of a number

\textit{pow()}, \textit{powf()}\quad Raise a number to a given power

\textit{sqrt()}, \textit{sqrtf()}\quad Calculate the nonnegative square root of a number

\textbf{Rounding}

\textit{ceil()}, \textit{ceilf()}\quad Round up a value to the next integer

\textit{floor()}, \textit{floorf()}\quad Round down a value to the next integer

\textit{rint()}, \textit{rintf()}\quad Round to the nearest integral value

\textbf{Trigonometric and hyperbolic functions}

\textit{acos()}, \textit{acosf()}\quad Compute the arccosine of an angle

\textit{acosh()}, \textit{acoshf()}\quad Compute the inverse hyperbolic cosine

\textit{asin()}, \textit{asinf()}\quad Compute the arcsine of an angle

\textit{asinh()}, \textit{asinhf()}\quad Compute the inverse hyperbolic sine

\textit{atan()}, \textit{atanf()}\quad Compute the arctangent of an angle

\textit{atanh()}, \textit{atanhf()}\quad Compute the inverse hyperbolic tangent

\textit{atan2()}, \textit{atan2f()}\quad Compute the arctangent, determining the quadrant
Summary of function categories

\[
\begin{align*}
\cos(), \cosf() & \quad \text{Compute the cosine of an angle} \\
\cosh(), \coshf() & \quad \text{Compute the hyperbolic cosine} \\
\sin(), \sinf() & \quad \text{Calculate the sine of an angle} \\
\sinh(), \sinhf() & \quad \text{Compute the hyperbolic sine} \\
\tan(), \tanf() & \quad \text{Calculate the tangent of an angle} \\
\tanh(), \tanhf() & \quad \text{Calculate the hyperbolic tangent}
\end{align*}
\]

Memory allocation functions

These functions allocate and deallocate blocks of memory:

\[
\begin{align*}
\text{alloca() } & \quad \text{Allocate automatic space from the stack} \\
_\text{ambilksz} & \quad \text{The increment for the break pointer} \\
_\text{btext} & \quad \text{The beginning of the text segment} \\
\text{calloc() } & \quad \text{Allocate space for an array} \\
\text{cfree() } & \quad \text{Free allocated memory} \\
_\text{edata} & \quad \text{The end of the data segment, excluding BSS data} \\
_\text{end} & \quad \text{The end of the data segment, including BSS data} \\
_\text{etext} & \quad \text{The end of the text segment} \\
\text{free() } & \quad \text{Deallocate a block of memory} \\
\text{ftw() } & \quad \text{Walk a file tree} \\
\text{longjmp() } & \quad \text{Restore the environment saved by setjmp()} \\
\text{mallinfo() } & \quad \text{Get memory allocation information} \\
\text{malloc() } & \quad \text{Allocate memory} \\
\text{mallopt() } & \quad \text{Control the memory allocation}
\end{align*}
\]
Summary of function categories

mcheck() Enable memory allocation routine consistency checks
memalign() Allocate aligned memory
mprobe() Perform consistency check on memory
posix_memalign() Allocate aligned memory
realloc() Allocate, reallocate or free a block of memory
sbrk() Set the allocation break value for a program
_scalloc() Allocate space for an array
setjmp() Save the calling environment, for use by longjmp()
siglongjmp() Restore the signal mask for a process, if one was saved
sigsetjmp() Save the environment, including the signal mask
_sfree() Deallocate a block of memory
_smalloc() Allocate memory in blocks
_srealloc() Allocate, reallocate or free a block of memory
valloc() Allocate a heap block aligned on a page boundary

Memory manipulation functions

These functions manipulate blocks of memory. In each case, the address of the memory block and its size is passed to the function.(Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.)

brk() Change the amount of space allocated for the calling process’s data segment
bzero() Set the first part of an object to null bytes
Summary of function categories

- `ffs()`  
  Find the first bit set in a bit string

- `index()`  
  Find a character in a string

- `memccpy()`  
  Copy bytes until a given character is found

- `memchr()`  
  Find the first occurrence of a character in a buffer

- `memcmp()`  
  Compare a given number of characters in two objects

- `memcpy()`  
  Copy a number of characters from one buffer to another

- `memcpyv()`  
  Copy a given number of structures

- `memcmp()`  
  Compare a given number of characters of two objects, without case sensitivity

- `mem_offset()`, `mem_offset64()`  
  Find offset of a mapped typed memory block

- `memmove()`  
  Copy bytes from one buffer to another, handling overlapping memory correctly

- `memset()`  
  Set the first part of an object to a given value

- `mlock()`  
  Lock a buffer in physical memory

- `mlockall()`  
  Lock a process’s address space

- `mmap()`, `mmap64()`  
  Map a memory region into a process address space

- `mmap_device_io()`  
  Gain access to a device’s registers

- `mmap_device_memory()`  
  Map a device’s physical memory into a process’s address space

- `msync()`  
  Synchronize memory with physical storage
Summary of function categories

munlock()       Unlock a buffer
munlockall()    Unlock a process’s address space
munmap_device_io()   Free access to a device’s registers
munmap_device_memory()  Unmap previously mapped addresses
posix_mem_offset(), posix_mem_offset64()  Find offset and length of a mapped typed memory block
rindex()        Find the last occurrence of a character in a string
shm_ctl()       Give special attributes to a shared memory object
shm_ctl_special()  Give special attributes to a shared memory object
swab()          Endian-swap a given number of bytes

See the section “String manipulation functions” for descriptions of functions that manipulate strings of data.

Message-queue functions

These functions deal with message queues:

mq_close()     Close a message queue
mq_getattr()   Get a message queue’s attributes
mq_notify()    Ask to be notified when there’s a message in the queue
mq_open()      Open a message queue
mq_receive()   Receive a message from a queue
mq_send() Send a message to a queue
mq_setattr() Set a message queue’s attributes
mq_timedreceive() Receive a message from a message queue
mq_timedsend() Send a message to a message queue
mq_unlink() Remove a message queue

Multibyte character functions

These ANSI C functions provide capabilities for processing multibyte characters. (Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.)

mblen() Count the bytes in a multibyte character
mbrlen() Count the bytes in a multibyte character (restartable)
mbrtowc() Convert a multibyte character into a wide character (restartable)
mbsinit() Determine the status of the conversion object used for restartable mb*() functions
mbsrtowcs() Convert a multibyte-character string into a wide-character string (restartable)
mbstowcs() Convert a multibyte-character string into a wide-character string
mbtowc() Convert a multibyte character into a wide character
QNX Neutrino-specific IPC functions

The following functions are defined:

- **ChannelCreate()**, **ChannelCreate r()**
  Create a communications channel

- **ChannelDestroy()**, **ChannelDestroy r()**
  Destroy a communications channel

- **_msg_info**
  Information about a message

- **MsgDeliverEvent()**, **MsgDeliverEvent r()**
  Deliver an event through a channel

- **MsgError()**, **MsgError r()**
  Unblock a client and set its errno

- **MsgInfo()**, **MsgInfo r()**
  Get additional information about a message

- **MsgKeyData()**, **MsgKeyData r()**
  Pass data through a common client

- **MsgRead()**, **MsgRead r()**
  Read data from a message

- **MsgReadv()**, **MsgReadv r()**
  Read data from a message

- **MsgReceive()**, **MsgReceive r()**
  Wait for a message or pulse on a channel

- **MsgReceivePulse()**, **MsgReceivePulse r()**
  Receive a pulse on a channel

- **MsgReceivePulsev()**, **MsgReceivePulsev r()**
  Receive a pulse on a channel
Summary of function categories

(MsgReceivev(), MsgReceivev_r())
Wait for a message or pulse on a channel

(MsgReply(), MsgReply_r())
Reply with a message

(MsgReplyv(), MsgReplyv_r())
Reply with a message

(MsgSend(), MsgSend_r())
Send a message to a channel

(MsgSendnc(), MsgSendnc_r())
Send a message to a channel (non-cancellation point)

(MsgSendPulse(), MsgSendPulse_r())
Send a pulse to a channel

(MsgSendsv(), MsgSendsv_r())
Send a message to a channel

(MsgSendsvnc(), MsgSendsvnc_r())
Send a message to a channel (non-cancellation point)

(MsgSendv(), MsgSendv_r())
Send a message to a channel

(MsgSendvc(), MsgSendvc_r())
Send a message to a channel (non-cancellation point)

(MsgSendvs(), MsgSendvs_r())
Send a message to a channel

(MsgSendvsnc(), MsgSendvsnc_r())
Send a message to a channel (non-cancellation point)
Summary of function categories

MsgVerifyEvent(), MsgVerifyEvent_r()
Check the validity of a receive ID and an event configuration

MsgWrite(), MsgWrite_r()
Write a reply

MsgWritev(), MsgWritev_r()
Write a reply

name_close()
Close the file descriptor returned by name_open()

name_open()
Open a name for a server connection

sigevent
Structure that describes an event

SyncTypeCreate(), SyncTypeCreate_r()
Create a synchronization object

Operating system I/O functions

These functions operate at the operating-system level, and are included for compatibility with other C implementations. For new programs, we recommended that you use the functions in the section “File manipulation functions”, functions are defined portably and are part of the ANSI standard for the C language.

The functions in this section reference opened files and devices using a file descriptor that’s returned when the file is opened. The file descriptor is passed to the other functions.

The following functions are defined:

chsize() Change the size of a file
cfgopen() Open a configuration file
close() Close a file at the operating system level
_cmdfd() Return a file descriptor for the executable file
Summary of function categories

(cmdname()) Find the path used to invoke the current process

(creat(), creat64()) Create and open a file at the operating system level

devctl() Control a device
dup() Duplicate a file descriptor
dup2() Duplicate a file descriptor, specifying the new descriptor
eof() Determine if the end-of-file has been reached

fcfgopen() Open a configuration file

fcntl() Provide control over an open file

fdatasync() Write queued file data to disk

fileno() Return the number of the file descriptor for a stream

flink() Assign a pathname to a file descriptor

flockfile() Acquire ownership of a file

fstat(), fstat64() Obtain information about an open file, given a file descriptor

fstatvfs(), fstatvfs64() Get filesystem information, given a file descriptor

fsync() Write queued file and filesystem data to disk

ftrylockfile() Acquire ownership of a file, without blocking

funlockfile() Release ownership of a file

GETIOVBASE() Get the base member of an iov_t structure

GETIOVLEN() Get the length member of an iov_t structure
Summary of function categories © 2007, QNX Software Systems GmbH & Co. KG.

getdtablesize()  Get the size of the file descriptor table
getrusage()  Get information about resource utilization
in8()  Read an 8-bit value from a port
in8s()  Read 8-bit values from a port
in16(), inbe16(), inle16()  
  Read a 16-bit value from a port
in16s()  Read 16-bit values from a port
in32(), inbe32(), inle32()  
  Read a 32-bit value from a port
in32s()  Read 32-bit values from a port
ioctl()  Control device
link()  Create a link to an existing file
lseek(), lseek64()  
  Set the current file position at the OS level
lio_listio()  Initiate a list of I/O requests
mknod()  Make a new filesystem entry point
modem_open()  Open a serial port
modem_read()  Read bytes from a file descriptor
modem_script()  Run a script on a device
modem_write()  Write a string to a device
name_close()  Close the file descriptor obtained with name_open()
name_open()  Open a name for a server connection
open(), open64()  
Open a file

openfd()  
Open for private access a file associated with a given descriptor

out8()  
Write an 8-bit value to a port

out8s()  
Write 8-bit values to a port

out16(), outbe16(), outle16()  
Write a 16-bit value to a port

out16s()  
Write 16-bit values to a port

out32(), outbe32(), outle32()  
Write a 32-bit value to a port

out32s()  
Write 32-bit values to a port

pathmgr_symlink()  
Create a symlink in the process manager

pathmgr_unlink()  
Remove the link created by pathmgr_symlink()

pipe()  
Create a pipe

poll()  
Multiplex input/output over a set of file descriptors

popen()  
Execute a command, creating a pipe to it

pread(), pread64()  
Read from a file without moving the file pointer

rdchk()  
Check to see if a read is likely to succeed

read()  
Read bytes from a file

readblock()  
Read blocks of data from a file
### Summary of function categories

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readcond()</td>
<td>Read data from a terminal device</td>
</tr>
<tr>
<td>readlink()</td>
<td>Place the contents of a symbolic link into a buffer</td>
</tr>
<tr>
<td>readv()</td>
<td>Read bytes from a file</td>
</tr>
<tr>
<td>select()</td>
<td>Check for files that are ready for reading or writing</td>
</tr>
<tr>
<td>SETIOV()</td>
<td>Fill in the fields of an iov_t structure</td>
</tr>
<tr>
<td>sopen()</td>
<td>Open a file for shared access</td>
</tr>
<tr>
<td>sopenfd()</td>
<td>Open for shared access a file associated with a given descriptor</td>
</tr>
<tr>
<td>symlink()</td>
<td>Create a symbolic link to a path</td>
</tr>
<tr>
<td>tcischars()</td>
<td>Determine the number of characters waiting to be read</td>
</tr>
<tr>
<td>tell(), tell64()</td>
<td>Determine the current file position</td>
</tr>
<tr>
<td>umask()</td>
<td>Set the file mode creation mask for the process</td>
</tr>
<tr>
<td>uname()</td>
<td>Get information about the operating system</td>
</tr>
<tr>
<td>unlink()</td>
<td>Delete a file</td>
</tr>
<tr>
<td>write()</td>
<td>Write bytes to a file</td>
</tr>
<tr>
<td>writeblock()</td>
<td>Write blocks of data to a file</td>
</tr>
<tr>
<td>writev()</td>
<td>Write bytes to a file</td>
</tr>
</tbody>
</table>

### PC Card functions

The following functions are defined:

- **pccard_arm()**: Arm the devp-pccard server
- **pccard_attach()**: Attach to the devp-pccard server
- **pccard_detach()**: Detach from the devp-pccard server
Summary of function categories

**pccard_info()**
Obtain socket information from the `devp-pccard` server

**pccard_lock()**
Lock the window of the card in the selected socket

**pccard_raw_read()**
Read the raw CIS data from the PC Card

**pccard_unlock()**
Unlock the window of the card in the selected socket

**Platform-specific functions**

These functions are for invoking Intel 80x86 and other processor-related functions directly from a program. Functions that apply to the Intel 8086 CPU apply to that family including the 80286, 80386, 80486 and Pentium processors.

You’ll also find endian-related functions listed here.

The following functions are defined:

**ENDIAN_BE16()**
Return a big-endian 16-bit value in native format

**ENDIAN_BE32()**
Return a big-endian 32-bit value in native format

**ENDIAN_BE64()**
Return a big-endian 64-bit value in native format

**ENDIAN_LE16()**
Return a little-endian 16-bit value in native format

**ENDIAN_LE32()**
Return a little-endian 32-bit value in native format

**ENDIAN_LE64()**
Return a little-endian 64-bit value in native format
**Summary of function categories**

© 2007, QNX Software Systems GmbH & Co. KG.

```
ENDIAN_RET16()
Return an endian-swapped 16-bit value

ENDIAN_RET32()
Return an endian-swapped 32-bit value

ENDIAN_RET64()
Return an endian-swapped 64-bit value

ENDIAN_SWAP16()
Endian-swap a 16-bit value in place

ENDIAN_SWAP32()
Endian-swap a 32-bit value in place

ENDIAN_SWAP64()
Endian-swap a 64-bit value in place

_intr_v86()  Execute a real-mode software interrupt

offsetof()  Return the offset of an element within a structure

sysmgr_reboot()  Reboot a QNX Neutrino system
```

**Process environment functions**

These functions deal with process identification, user identification, process groups, system identification, system time and process time, environment variables, terminal identification, and configurable system variables:

```
_argc  The number of arguments passed to main()

_argv  A pointer to the vector of arguments passed to main()

_auxv  A pointer to a vector of auxiliary arguments to main()
```
Summary of function categories

clearenv()  Clear the process environment area
confstr()   Get configuration-defined string values
ctermd()    Generate the pathname of the current controlling terminal
endutent()  Close the current user-information file
environ     Pointer to the process’s environment variables
err(), errx() Display a formatted error message, and then exit
errno       Global error variable
getegid()   Get the effective group ID
getenv()    Get the value of an environment variable
geteuid()   Get the effective user ID
getgid()    Get the group ID
getgrouplist() Determine the group access list for a user
getgroups() Get the supplementary group IDs of the calling process
getlogin()  Get the user name associated with the calling process
getlogin_r() Get the user name associated with the calling process
getopt()    Parse options from a command line
getpgid()   Get a process’s group ID
getpgrp()   Get the process group
getpid()    Get the process ID
getppid()   Get the parent process ID
### Summary of Function Categories

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getsid()</code></td>
<td>Get the session ID of a process</td>
</tr>
<tr>
<td><code>getuid()</code></td>
<td>Get the user ID</td>
</tr>
<tr>
<td><code>getutent()</code></td>
<td>Read the next entry from the user-information file</td>
</tr>
<tr>
<td><code>getutid()</code></td>
<td>Search for an entry in the user-information file</td>
</tr>
<tr>
<td><code>getutline()</code></td>
<td>Get an entry from the user-information file</td>
</tr>
<tr>
<td><code>initgroups()</code></td>
<td>Initialize the supplementary group access list</td>
</tr>
<tr>
<td><code>isatty()</code></td>
<td>Test to see if a file descriptor is associated with a terminal</td>
</tr>
<tr>
<td><code>login_tty()</code></td>
<td>Prepare for a login in a tty</td>
</tr>
<tr>
<td><code>main()</code></td>
<td>The function where program execution begins</td>
</tr>
<tr>
<td><code>ND_NODE_CMP()</code></td>
<td>Compare two node descriptor IDs</td>
</tr>
<tr>
<td><code>netmgr_ndtostr()</code></td>
<td>Convert a node descriptor into a string</td>
</tr>
<tr>
<td><code>netmgr_remote_nd()</code></td>
<td>Get a node descriptor that’s relative to a remote node</td>
</tr>
<tr>
<td><code>netmgr_strtond()</code></td>
<td>Convert a string into a node descriptor</td>
</tr>
<tr>
<td><code>__progname</code></td>
<td>The basename of the program being executed</td>
</tr>
<tr>
<td><code>putenv()</code></td>
<td>Add, change, or delete an environment variable</td>
</tr>
<tr>
<td><code>pututline()</code></td>
<td>Write an entry in the user-information file</td>
</tr>
<tr>
<td><code>searchenv()</code></td>
<td>Search the directories listed in an environment variable</td>
</tr>
<tr>
<td><code>setegid()</code></td>
<td>Set the effective group ID for a process</td>
</tr>
</tbody>
</table>
setenv() Set one or more environment variables
seteuid() Set the effective user ID
setgid() Set the real, effective and saved group IDs
setgroups() Set supplementary group IDs
setlocale() Set a program’s locale.
setpgid() Join or create a process group
setpgrp() Set the process group
setregid() Set real and effective group IDs
setreuid() Set real and effect user IDs
setsid() Create a new session
setuid() Set the real, effective and saved user IDs
setutent() Return to the beginning of the user-information file
strerror() Convert an error number into an error message
sysconf() Return the value of a configurable system limit
ttyname() Get a fully qualified pathname for a file
ttyname_r() Get a fully qualified pathname for a file
unsetenv() Remove an environment variable
utmp Entry in a user-information file
utmpname() Change the name of the user-information file
verr(), verrx() Display a formatted error message, and then exit (varargs)
warn(), warnx() Formatted error message (varargs)
warn() Formatted error message
Process manipulation functions

These functions deal with: process creation, execution, and termination; signal handling; and timer operations.

When you start a new process, it replaces the existing process if:

- You specify P_OVERLAY when calling one of the spawn* functions.
- You call one of the exec* routines.

The existing process may be suspended while the new process executes (control continues at the point following the place where the new process was started) in the following situations:

- You specify P_WAIT when calling one of the spawn* functions.
- You call system().

The following functions are defined:

- **abort()**
  Raise the SIGABRT signal to terminate program execution

- **alarm()**
  Schedule an alarm

- **assert()**
  Print a diagnostic message and optionally terminate the program

- **atexit()**
  Register functions to be called when the program terminates normally

- **ConnectAttach(), ConnectAttach_r()**
  Establish a connection between a process and a channel

- **ConnectClientInfo(), ConnectClientInfo_r()**
  Store information about a client connection

- **ConnectDetach(), ConnectDetach_r()**
  Break a connection between a process and a channel
ConnectFlags(), ConnectFlags_r()
Modify the flags associated with a connection

ConnectServerInfo(), ConnectServerInfo_r()
Store information about a connection

daemon()
Run a program in the background

DebugBreak()
Enter the process debugger

DebugKDBreak()
Enter the kernel debugger

DebugKDOoutput()
Print text with the kernel debugger

delay()
Suspend a process for a given length of time

dladdr()
Translate an address to symbolic information

dlclose()
Close a shared object

dlerror()
Get dynamic loading diagnostic information

dlopen()
Gain access to an executable object file

dlsect()
Get the address of a symbol in a shared object

execl()
Execute a file

execle()
Execute a file

execlp()
Execute a file

execlepe()
Execute a file

execv()
Execute a file

execve()
Execute a file

execvp()
Execute a file

execvpe()
Execute a file
Summary of function categories

_exit()  Terminate the program

exit()  Terminate the program

fork()  Create a new process

forkpty()  Create a new process operating in a pseudo-tty

getrlimit(), getrlimit64()  Get the limit on a system resource

getprio()  Get the priority of a given process

InterruptAttach(), InterruptAttach_r()  Attach an interrupt handler to an interrupt source

InterruptAttachEvent(), InterruptAttachEvent_r()  Attach an event to an interrupt source

InterruptDetach(), InterruptDetach_r()  Detach an interrupt handler by ID

InterruptDisable()  Disable hardware interrupts

InterruptEnable()  Enable hardware interrupts

InterruptHookIdle()  Attach an “idle” interrupt handler

InterruptHookTrace()  Attach the pseudo interrupt handler that the instrumented module uses

InterruptLock()  Protect critical sections of an interrupt handler

InterruptMask()  Disable a hardware interrupt
InterruptUnlock()  
Release a critical section locked with InterruptLock()  

InterruptUnmask()  
Enable a hardware interrupt  

InterruptWait(), InterruptWait_r()  
Wait for a hardware interrupt  

_intr_v86()  
Execute a real-mode software interrupt  

kill()  
Send a signal to a process or a group of processes  

killpg()  
Send a signal to a process group  

nap()  
Sleep for a given number of milliseconds  

napms()  
Sleep for a given number of milliseconds  

nice()  
Change the priority of a process  

openpty()  
Find an available pseudo-tty  

procmgr_daemon()  
Run a process in the background  

procmgr_event_notify()  
Ask to be notified of system-wide events  

procmgr_event_trigger()  
Trigger a global system event  

procmgr_guardian()  
Let a daemon process takeover as parent = guardian  

procmgr_session()  
Provide process manager session support  

raise()  
Signal an exceptional condition
Summary of function categories

\( \text{SchedCtl()} \) Control the adaptive partitioning scheduler

\( \text{SchedGet()}, \text{SchedGet}_r() \)
Get the scheduling policy for a thread

\( \text{SchedInfo()}, \text{SchedInfo}_r() \)
Get scheduler information

\( \text{SchedSet()}, \text{SchedSet}_r() \)
Set the scheduling policy for a thread

\( \text{SchedYield()}, \text{SchedYield}_r() \)
Yield to other threads

\( \text{setitimer()} \)
Set the value of an interval timer

\( \text{setprio()} \)
Set the priority of a process

\( \text{setrlimit()}, \text{setrlimit64()} \)
Set the limit on a system resource

\( \text{sigaction()} \)
Examine or specify the action associated with a signal

\( \text{sigaddset()} \)
Add a signal to a set

\( \text{sigblock()} \)
Add to the mask of signals to block

\( \text{sigdelset()} \)
Delete a signal from a set

\( \text{sigemptyset()} \)
Initialize a set to contain no signals

\( \text{sigfillset()} \)
Initialize a set to contain all signals

\( \text{sigismember()} \)
See if a given signal is in a given set

\( \text{sigmask()} \)
Construct a mask for a signal number

\( \text{signal()} \)
Set handling for exceptional conditions

\( \text{SignalAction()}, \text{SignalAction}_r() \)
Examine and/or specify actions for signals
SignalKill(), SignalKill_r()
    Send a signal to a process group, process or thread

SignalProcmask(), SignalProcmask_r()
    Modify or examine the signal blocked mask of a thread

SignalSuspend(), SignalSuspend_r()
    Suspend a process until a signal is received

SignalWaitinfo(), SignalWaitinfo_r()
    Select a pending signal

sigpause()    Wait for a signal

sigpending()   Examine the set of pending, masked signals for a process

sigprocmask()  Examine or change the signal mask for a process

sigqueue()     Queue a signal to a process

sigsetmask()   Set the mask of signals to block

sigsuspend()  Replace the signal mask, and then suspend the process

sigtimedwait() Wait for a signal or a timeout

signunblock()  Unblock signals

sigwait()      Wait for a pending signal

sigwaitinfo()  Wait for a pending signal and get its information

sleep()        Suspend a process for a given length of time

spawn()        Create and execute a new child process

spawnl()       Create and execute a new child process

spawnle()      Create and execute a new child process
Summary of functions categories

spawnlp()  Create and execute a new child process
spawnlpe() Create and execute a new child process
spawnp()  Create and execute a new child process
spawnv()  Create and execute a new child process
spawnve() Create and execute a new child process
spawnvp() Create and execute a new child process
spawnvpe() Create and execute a new child process

SyncCondvarSignal(), SyncCondvarSignal_r()  Wake up any threads that are blocked on a synchronization object

SyncCondvarWait(), SyncCondvarWait_r()  Block a thread on a synchronization object

SyncCtl(), SyncCtl_r()  Perform an operation on a synchronization object

SyncDestroy(), SyncDestroy_r()  Destroy a synchronization object

SyncMutexEvent(), SyncMutexEvent_r()  Attach an event to a mutex

SyncMutexLock(), SyncMutexLock_r()  Lock a mutex synchronization object

SyncMutexUnlock(), SyncMutexUnlock_r()  Unlock a mutex synchronization object

SyncMutexRevive(), SyncMutexRevive_r()  Revive a mutex that’s in the DEAD state

SyncSemPost(), SyncSemPost_r()  Increment a semaphore
Summary of function categories

`SyncSemWait()`, `SyncSemWait_r()`  
Wait on a semaphore

`system()`  
Execute a system command

`SYSPAGE_CPU_ENTRY()`  
Return a CPU-specific entry from the system page

`SYSPAGE_ENTRY()`  
Return an entry from the system page

`_syspage_ptr`  
A pointer to the system page

`ThreadCancel()`, `ThreadCancel_r()`  
Cancel a thread

`ThreadCreate()`, `ThreadCreate_r()`  
Create a new thread

`ThreadCtl()`, `ThreadCtl_r()`  
Control a thread

`ThreadDestroy()`, `ThreadDestroy_r()`  
Destroy a thread immediately

`ThreadDetach()`, `ThreadDetach_r()`  
Detach a thread from a process

`ThreadJoin()`, `ThreadJoin_r()`  
Block until a thread terminates

`TraceEvent()`  
Trace kernel events

`ualarm()`  
Schedule an alarm

`usleep()`  
Suspend a thread for a given number of microseconds

`vfork()`  
Spawn a new process and block the parent

`wait()`  
Wait for the status of a terminated child process
Summary of function categories  © 2007, QNX Software Systems GmbH & Co. KG.

wait3()   Wait for a child process to change state  
wait4()   Wait for a child process to terminate or stop  
waitid()   Wait for a child process to change state  
waitpid()   Suspend the calling process  

There are eight spawn*() and exec*() functions each. The * is one to three letters, where:

- 1 or v (one is required) indicates the way the process parameters are passed  
- p (optional) indicates that the PATH environment variable is searched to locate the program for the process  
- e (optional) indicates that the environment variables are being passed  

Realtime timer functions

These functions provide realtime timer capabilities:

clock_gettime() Get the current time of a clock  
clock_gettime() Set a clock  

clock_gettime() Get the value of an interval timer  
nanosleep() Suspend process until a timeout or signal occurs  
nanospin() Busy-wait without thread blocking for a period of time  
nanospin_calibrate() Calibrate before calling nanospin*()
Summary of function categories

nanospin_count()

Busy-wait without blocking for a number of iterations

nanospin_ns()

Busy-wait without blocking for a period of time

nanospin_ns_to_count()

Convert a time in nanoseconds into a number of iterations

sched_getparam()

Get the current priority of a process

sched_get_priority_adjust()

Calculate the allowable priority for the scheduling policy

sched_get_priority_max()

Get the maximum value for the scheduling policy

sched_get_priority_min()

Get the minimum value for the scheduling policy

sched_getscheduler()

Get the current scheduling policy for a process

sched_param

Structure that describes scheduling parameters

sched_rr_get_interval()

Get the execution time limit of a process

sched_setparam()

Change the priority of a process

sched_setscheduler()

Change the priority and scheduling policy of a process
**Resource manager functions**

These functions help you create resource managers. For an overview of these functions, see “Components of a resource manager” in the Writing a Resource Manager chapter of the QNX Neutrino Programmer’s Guide.

- `_io_connect` Structure of a resource manager’s connect message
- `_io_connect_ftype_reply` Structure of a connect message giving a status and a file type
- `_io_connect_link_reply` Structure of a connect message that redirects a client to another resource
- `iofdinfo()` Retrieve server attributes
- `iofunc_attr_init()` Initialize the default attribute structure
Summary of function categories

`iofunc_attr_lock()`
Lock the attribute structure

`iofunc_attr_t`
Attribute structure

`iofunc_attr_trylock()`
Try to lock the attribute structure

`iofunc_attr_unlock()`
Unlock the attribute structure

`iofunc_check_access()`
Check access permissions

`iofunc_chmod()` Handle an _IO_CHMOD message

`iofunc_chmod_default()`
Default handler for _IO_CHMOD messages

`iofunc_chown()` Handle an _IO_CHOWN message

`iofunc_chown_default()`
Default handler for _IO_CHOWN messages

`iofunc_client_info()`
Return information about a client connection

`iofunc_close_dup()`
Frees all locks allocated for the client process

`iofunc_close_dup_default()`
Default handler for _IO_CLOSE messages

`iofunc_close_ocb()`
Return the memory allocated for an OCB

`iofunc_close_ocb_default()`
Return the memory allocated for an OCB
Summary of function categories

iofunc_devctl() Handle an _IO_DEVCTL message

iofunc_devctl_default() Default handler for _IO_DEVCTL messages

iofunc_fdinfo() Handle an _IO_FINFO message

iofunc_fdinfo_default() Default handler for _IO_FINFO messages

iofunc_func_init() Initialize the default POSIX-layer function tables

iofunc_link() Link two directories

iofunc_lock() Lock a resource

iofunc_lock_alloc() Allocate memory to lock structures

iofunc_lock_default() Default handler for _IO_LOCK messages

iofunc_lock_free() Return memory allocated for lock structures

iofunc_lock_ocb_default() Default handler for the lock_ocb callout

iofunc_lseek() Handle an _IO_LSEEK message

iofunc_lseek_default() Default handler for _IO_LSEEK message

iofunc_mknod() Verify a client’s ability to make a new filesystem entry point

iofunc_mmap() Handle an _IO_MMAP message
Summary of function categories

```c
iofunc_mmap_default()
    Default handler for IO_MMAP messages

iofunc_notify()
    Install, poll, or remove a notification handler

iofunc_notify_remove()
    Remove notification entries from list

iofunc_notify_trigger()
    Send notifications to queued clients

iofunc_ocb_attach()
    Initialize an Open Control Block

iofunc_ocb_calloc()
    Allocate an iofunc OCB

iofunc_ocb_detach()
    Release OCB resources

iofunc_ocb_free()
    Deallocate an iofunc OCBs memory

iofunc_ocb_t
    Open Control Block structure

iofunc_open()
    Verify a client’s ability to open a resource

iofunc_open_default()
    Default handler for _IO_CONNECT messages

iofunc_openfd()
    Increment count and locking flags

iofunc_openfd_default()
    Default handler for _IO_OPENFD messages

iofunc_pathconf()
    Support pathconf() requests

iofunc_pathconf_default()
    Default handler for _IO_PATHCONF messages
```
Summary of function categories

\textit{iofunc\_read\_default()}

Default handler for _IO_READ messages

\textit{iofunc\_readlink()}

Verify a client’s ability to read a symbolic link

\textit{iofunc\_read\_verify()}

Verify a client’s read access to a resource

\textit{iofunc\_rename()}

Do permission checks for a _IO_CONNECT_rename message

\textit{iofunc\_space\_verify()}

Do permission checks for _IO_SPACE message

\textit{iofunc\_stat()}

Populate a \textbf{stat} structure

\textit{iofunc\_stat\_default()}

Default handler for _IO_STAT messages

\textit{iofunc\_sync()}

Indicate if synchronization is needed

\textit{iofunc\_sync\_default()}

Default handler for _IO_SYNC messages

\textit{iofunc\_sync\_verify()}

Verify permissions to sync

\textit{iofunc\_time\_update()}

Update time stamps

\textit{iofunc\_unblock()}

Unblock OCBs

\textit{iofunc\_unblock\_default()}

Default unblock handler

\textit{iofunc\_unlink()}

Verify that an entry can be unlinked
Summary of function categories

*iocfunc_unlock_ocb_default()*

Default handler for the *unlock_ocb* callout

*iocfunc_utime()*

Update time stamps

*iocfunc_utime_default()*

Default handler for _IO_UTIME messages

*iocfunc_write_default()*

Default handler for _IO_WRITE messages

*iocfunc_write_verify()*

Verify a client’s write access to a resource

*ionotify()*

Arm a resource manager

*mount()*

Mount a filesystem

*mount_parse_generic_args()*

Strip off common mount arguments

*resmgr_devino()*

Get the device and inode number

*resmgr_open_bind()*

Associate an OCB with a process

*rsrdbmgr_attach()*

Reserve a system resource for a process

*rsrdbmgr_create()*

Create a system resource

*rsrdbmgr_destroy()*

Destroy a system resource

*rsrdbmgr_detach()*

Return a system resource to the resource database
Summary of function categories

\textit{rsrdbmgr\_devno\_attach()}
\begin{itemize}
  \item Get a major and minor number
\end{itemize}

\textit{rsrdbmgr\_devno\_detach()}
\begin{itemize}
  \item Detach a major and minor number
\end{itemize}

\textit{rsrdbmgr\_query()}
\begin{itemize}
  \item Query the resource database
\end{itemize}

\textit{umount()}
\begin{itemize}
  \item Unmount a filesystem
\end{itemize}

\section*{Searching and sorting functions}

These functions provide searching and sorting capabilities (Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.):

\textit{alphasort()}
\begin{itemize}
  \item Compare two directory entries
\end{itemize}

\textit{bsearch()}
\begin{itemize}
  \item Perform a binary search on a sorted array
\end{itemize}

\textit{ffs()}
\begin{itemize}
  \item Find the first bit set in a bit string
\end{itemize}

\textit{hcreate()}
\begin{itemize}
  \item Create a hash search table
\end{itemize}

\textit{hdestroy()}
\begin{itemize}
  \item Destroy the hash search table
\end{itemize}

\textit{hsearch()}
\begin{itemize}
  \item Search the hash search table
\end{itemize}

\textit{index()}
\begin{itemize}
  \item Find a character in a string
\end{itemize}

\textit{lfind()}
\begin{itemize}
  \item Find entry in a linear search table
\end{itemize}

\textit{lsearch()}
\begin{itemize}
  \item Linear search and update
\end{itemize}

\textit{pathfind(), pathfind\_r()}
\begin{itemize}
  \item Search for a file in a list of directories
\end{itemize}

\textit{qsort()}
\begin{itemize}
  \item Sort an array, using a modified Quicksort algorithm
\end{itemize}

\textit{re\_comp()}
\begin{itemize}
  \item Compile a regular expression
\end{itemize}
Summary of function categories

\texttt{re\_exec()} \quad \text{Execute a regular expression}

\texttt{regcomp()} \quad \text{Compile a regular expression}

\texttt{regerror()} \quad \text{Explain a regular expression error code}

\texttt{regexec()} \quad \text{Compare a string with a compiled regular expression}

\texttt{regfree()} \quad \text{Release memory allocated for a regular expression}

\texttt{rindex()} \quad \text{Find a character in a string}

\texttt{strcspn()} \quad \text{Count the characters at the beginning of a string that aren’t in a character set}

\texttt{strstr()} \quad \text{Find one string inside another}

\section*{Shared memory functions}

These functions provide memory mapping capabilities:

\texttt{mmap()}, \texttt{mmap64()}
\quad \text{Map a memory region into a process address space}

\texttt{mprotect()} \quad \text{Change memory protection}

\texttt{munmap()} \quad \text{Unmap previously mapped addresses}

\texttt{shm\_ctl()} \quad \text{Give special attributes to a shared memory object}

\texttt{shm\_open()} \quad \text{Open a shared memory object}

\texttt{shm\_unlink()} \quad \text{Remove a shared memory object}

\section*{Signal functions}

These functions deal with handling and sending signals.

\texttt{DebugBreak()} \quad \text{Enter the process debugger}
Summary of function categories

DebugKDBreak()
- Enter the kernel debugger

DebugKDOoutput()
- Print text with the kernel debugger

kill()
- Send a signal to a process or a group of processes

killpg()
- Send a signal to a process group

pause()
- Suspend the calling thread until delivery of a signal

raise()
- Signal an exceptional condition

sigaction()
- Examine or specify the action associated with a signal

sigaddset()
- Add a signal to a set

sigdelset()
- Delete a signal from a set

sigemptyset()
- Initialize a set to contain no signals

sigfillset()
- Initialize a set to contain all signals

sigismember()
- See if a given signal is in a given set

signal()
- Set handling for exceptional conditions

SignalAction(), SignalAction_r()
- Examine and/or specify actions for signals

SignalKill(), SignalKill_r()
- Send a signal to a process group, process, or thread

SignalProcmask(), SignalProcmask_r()
- Modify or examine the signal blocked mask of a thread
Summary of function categories

SignalSuspend(), SignalSuspend_r()
Suspend a process until a signal is received

SignalWaitinfo(), SignalWaitinfo_r()
Select a pending signal

sigpending() Examine the set of pending, masked signals for a process

sigprocmask() Examine or change the signal mask for a process

sigqueue() Queue a signal to a process

sigsuspend() Replace the signal mask, and then suspend the process

sigtimedwait() Wait for a signal or a timeout

sigwait() Wait for a pending signal

sigwaitinfo() Wait for a pending signal and get its information

strsignal() Return the description of a signal

Stream I/O functions

A stream is the name given to a file or device that has been opened for data transmission. When a stream is opened, a pointer to a FILE structure is returned. This pointer is used to reference the stream when other functions are subsequently invoked.

When a program begins execution, a number of streams are already open for use:

stderr Standard Error: output to the console (used for error messages)

stdin Standard Input: input from the console

stdout Standard Output: output to the console
You can redirect these standard streams by calling `freopen()`.  

See also the section “File manipulation functions” for other functions that operate on files.  

The functions in the section “Operating system I/O functions” may also be invoked (use the `fileno()` function to get the file descriptor). Since the stream functions may buffer input and output, use these functions with caution to avoid unexpected results.  

(Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clearerr()</code></td>
<td>Clear the end-of-file and error indicators for a stream</td>
</tr>
<tr>
<td><code>fclose()</code></td>
<td>Close a stream</td>
</tr>
<tr>
<td><code>fcloseall()</code></td>
<td>Close all open stream files, except <code>stdin</code>, <code>stdout</code> and <code>stderr</code></td>
</tr>
<tr>
<td><code>fdopen()</code></td>
<td>Associate a stream with a file descriptor</td>
</tr>
<tr>
<td><code>feof()</code></td>
<td>Test the end-of-file indicator</td>
</tr>
<tr>
<td><code>ferror()</code></td>
<td>Test the error indicator for a stream</td>
</tr>
<tr>
<td><code>fflush()</code></td>
<td>Flush the input or output buffer for a stream</td>
</tr>
<tr>
<td><code>fgetc()</code></td>
<td>Get the next character from a file stream</td>
</tr>
<tr>
<td><code>fgetchar()</code></td>
<td>Get a character from <code>stdin</code></td>
</tr>
<tr>
<td><code>fgetpos()</code></td>
<td>Store the current position of a stream</td>
</tr>
<tr>
<td><code>fgets()</code></td>
<td>Get a string of characters from a stream</td>
</tr>
<tr>
<td><code>flushall()</code></td>
<td>Clear all input buffers and write all output buffers</td>
</tr>
<tr>
<td><code>fopen()</code></td>
<td>Open a stream</td>
</tr>
<tr>
<td><code>fprintf()</code></td>
<td>Write output to a stream</td>
</tr>
<tr>
<td><code>fputc()</code></td>
<td>Write a character to an output stream</td>
</tr>
</tbody>
</table>
Summary of function categories

- **fputchar()**: Write a character to `stdout`
- **fputs()**: Write a character string to an output stream
- **fread()**: Read elements of a given size from a stream
- **freopen()**: Reopen a stream
- **fscanf()**: Scan input from a stream
- **fseek(), fseeko()**: Change the read/write position of a stream
- **fsetpos()**: Set the current stream position
- **ftell(), ftello()**: Return the current read/write position of a stream
- **fwrite()**: Write a number of elements into a stream
- **getc()**: Get the next character from a stream
- **getchar()**: Get a character from `stdin`
- **getchar_unlocked()**: Get a character from `stdin`
- **getc_unlocked()**: Get the next character from a stream
- **gets()**: Get a string of characters from a stream
- **getw()**: Get a word from a stream
- **perror()**: Print, in `stderr`, the message associated with the value of `errno`
- **printf()**: Write formatted output to `stdout`
- **putc()**: Write a character to an output stream
- **putchar()**: Write a character to `stdout`
- **putchar_unlocked()**: Write a character to `stdout`
putc_unlocked()  
Write a character to an output stream

puts()  
Write a string to stdout

putw()  
Put a word on a stream

rewind()  
Set the file position indicator to the beginning of the stream

scanf()  
Scan formatted input from a stream

setbuf()  
Associate a buffer with a stream

setbuffer()  
Assign block buffering to a stream

setlinebuf()  
Assign line buffering to a stream

setvbuf()  
Associate a buffer with a stream

snprintf()  
Write formatted output to a character array, up to a given max number of characters

tmpfile()  
Create a temporary binary file

ungetc()  
Push a character back onto an input stream

vfprintf()  
Write formatted output to a file stream (varargs)

vfscanf()  
Scan input from a file stream (varargs)

vprintf()  
Write formatted output to standard output (varargs)

vscanf()  
Scan input from standard input (varargs)

See the section “Directory functions” for functions that are related to directories.
String manipulation functions

A string is an array of characters (with type char) that’s terminated with an extra null character (\0). Functions are passed only the address of the string, since the size can be determined by searching for the terminating character. (Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.)

**basename()**  
Find the part of a string after the last slash (/)

**bcmp()**  
Compare a given number of characters in two strings

**bcopy()**  
Copy a number of characters in one string to another

**fnmatch()**  
Check to see if a file or path name matches a pattern

**getsubopt()**  
Parse suboptions from a string

**index()**  
Find a character in a string

**input_line()**  
Get a string of characters from a file

**localeconv()**  
Set numeric formatting according to the current locale

**re_comp()**  
Compile a regular expression

**re_exec()**  
Execute a regular expression

**regcomp()**  
Compile a regular expression

**regerror()**  
Explain a regular expression error code

**regexec()**  
Compare a string with a compiled regular expression

**regfree()**  
Release memory allocated for a regular expression
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rindex()</td>
<td>Find a character in a string</td>
</tr>
<tr>
<td>sprintf()</td>
<td>Print formatted output into a string</td>
</tr>
<tr>
<td>sscanf()</td>
<td>Scan input from a character string</td>
</tr>
<tr>
<td>straddstr()</td>
<td>Concatenate one string on to the end of another</td>
</tr>
<tr>
<td>strcasecmp()</td>
<td>Compare two strings, ignoring case</td>
</tr>
<tr>
<td>strcat()</td>
<td>Concatenate two strings</td>
</tr>
<tr>
<td>strchr()</td>
<td>Find the first occurrence of a character in a string</td>
</tr>
<tr>
<td>strcmp()</td>
<td>Compare two strings</td>
</tr>
<tr>
<td>strcmpi()</td>
<td>Compare two strings, ignoring case</td>
</tr>
<tr>
<td>strcoll()</td>
<td>Compare two strings, using the locale’s collating sequence</td>
</tr>
<tr>
<td>strcpy()</td>
<td>Copy a string</td>
</tr>
<tr>
<td>strcsn()</td>
<td>Count the characters at the beginning of a string that aren’t in a given character set</td>
</tr>
<tr>
<td>strdup()</td>
<td>Create a duplicate of a string</td>
</tr>
<tr>
<td>strerror()</td>
<td>Map an error number to an error message</td>
</tr>
<tr>
<td>stricmp()</td>
<td>Compare two strings, ignoring case</td>
</tr>
<tr>
<td>strlen()</td>
<td>Compute the length of a string</td>
</tr>
<tr>
<td>strlwr()</td>
<td>Convert a string to lowercase</td>
</tr>
<tr>
<td>strncasecmp()</td>
<td>Compare two strings, ignoring case, up to a given length</td>
</tr>
<tr>
<td>strncat()</td>
<td>Concatenate two strings, up to a maximum length</td>
</tr>
<tr>
<td>strncmp()</td>
<td>Compare two strings, up to a given length</td>
</tr>
<tr>
<td>strncpy()</td>
<td>Copy a string, to a maximum length</td>
</tr>
</tbody>
</table>
strnicmp()  Compare two strings up to a given length, ignoring case

strnset()  Fill a string with a given character, to a given length

strpbrk()  Find the first character in a string that’s in a given character set

strrchr()  Find the last occurrence of a character in a string

strrev()  Reverse a string

strsep()  Separate a string into pieces marked by given delimiters

strset()  Fill a string with a given character

strspn()  Count the characters at the beginning of a string that are in a given character set

strstr()  Find one string inside another

strtok()  Break a string into tokens

strtok_r()  Break a string into tokens (reentrant)

strupr()  Convert a string to uppercase

strxfrm()  Transform one string into another, to a given length

vprintf()  Write formatted output to a buffer (varargs)

vsnprintf()  Write formatted output to a character array, up to a given max number of characters (varargs)

vsscanf()  Scan input from a string (varargs)

For related functions see these sections:

- “Conversion functions” — conversions to and from strings
Summary of function categories

• “Time functions” — formatting of dates and times
• “Memory manipulation functions” — operating on arrays without a terminating NUL character.

System database functions

The following functions are defined:

- `crypt()` Encrypt a password
- `encrypt()` Encrypt a string
- `endgrent()` Close the group database file
- `endpwent()` Close the password database file
- `endspent()` Close the shadow password database file
- `fgetspent()` Get an entry from the shadow password database
- `getgrent()` Return an entry from the group database
- `getgrgid()` Get information about the group with a given ID
- `getgrgid_r()` Get information about the group with a given ID
- `getgrnam()` Get information about the group with a given name
- `getgrnam_r()` Get information about the group with a given name
- `getpass()` Prompt for and read a password
- `getpwent()` Get an entry from the password database
- `getpwnam()` Get information about the user with a given name
- `getpwnam_r()` Get information about the user with a given name
- `getpwuid()` Get information about the user with a given ID
- `getpwuid_r()` Get information about the user with a given ID
getspent(), getspent_r()

Get an entry from the shadow password database

getspnam(), getspnam_r()

Get information about a user with a given name

putspent()

Put an entry into the shadow password database

qnx_crypt()

Encrypt a password (QNX 4)

setkey()

Set the key used in encryption

setgrent()

Rewind to the start of the group database file

setpwent()

Rewind the password database file

setspent()

Rewind the shadow password database file

System message log functions

The following functions are defined:

closelog()

Close the system log

openlog()

Open the system log

setlogmask()

Set the system log priority mask

slogb()

Send a message to the system logger

slogf()

Send a formatted message to the system logger

slogi()

Send a message to the system logger

syslog()

Write a message to the system log

vslogf()

Send a formatted message to the system logger (varargs)

vsyslog()

Control system log (varargs)
TCP/IP functions

These functions, prototypes and structures deal with TCP/IP network communications, database files, and the data server.

- **accept()**: Accept a connection on a socket
- **addrinfo**: TCP/IP address information
- **bind()**: Bind a name to a socket
- **bindresvport()**: Bind a socket to a privileged IP port
- **connect()**: Initiate connection on a socket
- **dn_comp()**: Compress an Internet domain name
- **dn_expansion()**: Expand a compressed Internet domain name
- **ds_clear()**: Delete a data server variable
- **ds_create()**: Create a data server variable
- **ds_deregister()**: Deregister an application with the data server
- **ds_flags()**: Set the flags for a data server variable
- **ds_get()**: Retrieve a data server variable
- **ds_register()**: Register an application with the data server
- **ds_set()**: Set a data server variable
- **endhostent()**: Close the TCP connection and the hosts file
- **endnetent()**: Close the network database
- **endprotoent()**: Close protocol name database file
- **endservent()**: Close network services database file
- **freeaddrinfo()**: Free an address information structure
- **freeifaddrs()**: Free an address information structure
Summary of function categories

- **gai_strerror()**  
  Return the string associated with a `getaddrinfo()` error code

- **getaddrinfo()**  
  Get address information

- **getdomainname()**  
  Get the domain name of the current host

- **gethostbyaddr()**  
  Get a network host entry, given an Internet address

- **gethostbyaddr_r()**  
  Get a network host entry, in a thread-safe manner

- **gethostbyname()**  
  Get a network host entry, given a name

- **gethostbyname2()**  
  Get a network host entry, given a name

- **gethostbyname_r()**  
  Get a network host entry by name

- **gethostent()**  
  Get the next entry from the host database

- **gethostent_r()**  
  Get the next entry from the host database

- **gethostname()**  
  Get the name of the current host

- **getifaddrs()**  
  Get a network interface address

- **getnameinfo()**  
  Perform address-to-nodename translation in a protocol-independent manner

- **getnetbyaddr()**  
  Get network entry

- **getnetbyname()**  
  Get network entry

- **getnetent()**  
  Get an entry from the network database

- **getpeername()**  
  Get name of connected peer

- **getprotobyname()**  
  Get protocol entry
getprotobynumber()  
Get protocol entry by number

getprotoent()  
Read next line of protocol name database file

getservbyname()  
Get service entry

getservbyport()  
Get service entry for a port

getservent()  
Read the next line of network services database file

getsockname()  
Get socket name

getsockopt()  
Get options on socket name

h_errno  
Host error variable

herror()  
Print the message associated with the value of h_errno to standard error

hostent  
Structure that describes an Internet host

hstrerror()  
Get an error message string associated with the error return status

htonl()  
Convert a 32-bit value from host-byte order to network-byte order

htons()  
Convert a 16-bit value from host-byte order to network-byte order

ICMP  
Internet Control Message Protocol

ICMP6  
Internet Control Message Protocol for IPv6

if_frenameindex()  
Free dynamic memory allocated by if_nameindex()

if_indextoname()  
Map an interface index to its name
Summary of function categories

- `if_nameindex()` Return a list of interfaces
- `if_nametoindex()` Map an interface name to its index
- `ifaddrs()` Structure that describes an Internet host
- `inet_addr()` Convert a string into an Internet address
- `inet_aton()` Convert a string into an Internet address
- `inet_lnaof()` Convert an Internet address into a local network address
- `inet_makeaddr()` Convert a network number and a local network address into an Internet address
- `inet_net_ntop()` Convert an Internet network number to CIDR format
- `inet_netof()` Convert Internet address into a network number
- `inet_netpton()` Convert an Internet network number from CIDR format to network format
- `inet_network()` Convert a string into an Internet network number
- `inet_ntoa()` Convert an Internet address into a string
- `inet_ntoa_r()` Convert an Internet address into a string
- `inet_ntop()` Convert a numeric network address to a string
- `inet_pton()` Convert a text host address to a numeric network address

INET6 Internet Protocol version 6 family

- `inet6_option_*()` Manipulate IPv6 hop-by-hop and destination options
- `inet6_rthdr_*()` Manipulate IPv6 Router header options
Summary of function categories

IP  Internet Protocol
IPsec  Internet security protocol

`ipsec_dump_policy()`  Generate readable string from IPsec policy specification

`ipsec_get_policylen()`  Get length of the IPsec policy

`ipsec_set_policy()`  Generate IPsec policy specification structure from readable string

`ipsec_strerror()`  Error code for IPsec policy manipulation library

IPv6  Internet Protocol version 6

`isfdtype()`  Determine whether a file descriptor refers to a socket

`listen()`  Listen for connections on a socket

`nbacconnect()`  Initiate a connection on a socket (nonblocking)

`nbacconnect_result()`  Get the status of the previous call to `nbacconnect()`

`netent`  Structure for information from the network database

`ntohl()`  Convert network-byte order value

`ntohs()`  Convert network-byte order value

`protoent`  Structure for information from the protocol database

`Raccept()`  Accept a connection on a socket (via a SOCKS server)
Summary of function categories

**Rbind()**
Bind a name to a socket (via a SOCKS server)

**rcmd()**
Execute a command on a remote host

**Rconnect()**
Initiate a connection on a socket (via a SOCKS server)

**read_main_config_file()**
Read the `snmpd.conf` file

**recv()**
Receive a message from a socket

**recvfrom()**
Receive a message from a socket

**recvmsg()**
Receive a message from a socket

**res_init()**
Initialize the Internet domain name resolver routines

**res_mkquery()**
Construct an Internet domain name query

**res_query()**
Make an Internet domain name query

**res_querydomain()**
Query the local Internet domain name server

**res_search()**
Make an Internet domain name search

**res_send()**
Send a preformatted Internet domain name query

**Rgetsockname()**
Get the name of a socket (via a SOCKS server)

**Rlisten()**
Listen for connections on a socket (via a SOCKS server)

**ROUTE**
System packet forwarding database

**Rrcmd()**
Execute a command on a remote host (via a SOCKS server)

**rresvport()**
Obtain a socket with a privileged address
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rselect()</td>
<td>Check for descriptors that are ready for reading or writing (via a SOCKS server)</td>
</tr>
<tr>
<td>ruserok()</td>
<td>Check the identity of a remote host</td>
</tr>
<tr>
<td>SCTP</td>
<td>Stream Control Transmission Protocol</td>
</tr>
<tr>
<td>sctp_bindx()</td>
<td>Add or remove one or more given addresses from an association</td>
</tr>
<tr>
<td>sctp_connectx()</td>
<td>Connect a host to a multihomed endpoint</td>
</tr>
<tr>
<td>sctp_freeladdr()</td>
<td>Free all resources allocated by sctp_getladdr()</td>
</tr>
<tr>
<td>sctp_freepadrs()</td>
<td>Free all resources allocated by sctp_getpadrs()</td>
</tr>
<tr>
<td>sctp_getladdr()</td>
<td>Get all locally bound addresses on a socket</td>
</tr>
<tr>
<td>sctp_getpadrs()</td>
<td>Get all peer addresses in an association</td>
</tr>
<tr>
<td>sctp_peeloff()</td>
<td>Branch off an association into a separate socket</td>
</tr>
<tr>
<td>sctp_recvmssg()</td>
<td>Receive a message, using advanced SCTP features</td>
</tr>
<tr>
<td>sctp_sendmsg()</td>
<td>Send a message, using advanced SCTP features</td>
</tr>
<tr>
<td>send()</td>
<td>Send a message to a socket</td>
</tr>
<tr>
<td>sendmsg()</td>
<td>Send a message to a socket</td>
</tr>
<tr>
<td>sendto()</td>
<td>Send a message to a socket</td>
</tr>
<tr>
<td>servant</td>
<td>Structure for information from the services database</td>
</tr>
<tr>
<td>setdomainname()</td>
<td>Set the domain name of the current host</td>
</tr>
<tr>
<td>sethostent()</td>
<td>Set the local hosts entry</td>
</tr>
<tr>
<td>sethostname()</td>
<td>Set the name of the current host</td>
</tr>
<tr>
<td>setnetent()</td>
<td>Open the network database</td>
</tr>
</tbody>
</table>
setprotoent()  Open protocol name database file
setservent()  Open network services database file
setsockopt()  Set options on socket name
shutdown()  Shut down part of a full-duplex connection
snmp_close()  Close an SNMP session

snmp_free_pdu()  Free an SNMP message structure

snmp_open()  Open an SNMP session

snmp_pdu  Structure that describes an SNMP Protocol Data Unit (transaction)

snmp_pdu_create()  Create an SNMP Protocol Data Unit message structure

snmp_read()  Read an SNMP message

snmp_select_info()  Get information that select() needs for SNMP

snmp_send()  Send SNMP messages

snmp_session  Structure that defines a set of transactions with similar transport characteristics

snmp_timeout()  Timeout during an SNMP session

sockatmark()  Determine whether a socket is at the out-of-band mark

socket()  Create an endpoint for communication

socketpair()  Create a pair of connected sockets or a bi-directional pipe
Summary of function categories

SOCKSinit() Initialize a connection with a SOCKS server
sysctl() Get or set the system information
TCP Internet Transmission Control Protocol
UDP Internet User Datagram Protocol
UNIX UNIX-domain protocol family

Terminal control functions

The following functions are defined:

cfgetispeed() Return the input baud rate that’s stored in a termios structure
cfgetospeed() Return the output baud rate that’s stored in a termios structure
cfmakeraw() Set terminal attributes
cfsetispeed() Set the input baud rate in a termios structure
cfsetospeed() Set the output baud rate in a termios structure
tcdrain() Wait until all output has been transmitted to a device
tcdropline() Disconnect a communications line
tcflow() Perform a flow-control operation on a data stream
tcflush() Flush the input and/or output stream
tcgetattr() Get the current terminal control settings
tcgetpgrp() Get the process group ID associated with a device
tcgetsid() Get the process group ID of the session leader for a controlling terminal
tcgetsize() Get the size of a character device
Summary of function categories

`tcinject()` Inject characters into a device's input buffer

`tcischars()` Determine the number of characters waiting to be read

`tcsendbreak()` Assert a break condition over a communications line

`tcsetattr()` Change the terminal control settings for a device

`tcsetpgrp()` Set the process group ID for a device

`tcsetsid()` Make a terminal device a controlling device

`tcsetsize()` Set the size of a character device

`termios` Terminal control structure

Thread functions

These functions deal with threads and the objects used to synchronize threads:

`pause()` Suspend the calling thread until delivery of a signal

`pthread_abort()` Unconditionally terminate the target thread

`pthread_atfork()` Register fork handlers

`pthread_attr_destroy()` Destroy the thread attribute object

`pthread_attr_getdetachstate()` Get the thread detach state attribute

`pthread_attr_getguardsize()` Get the thread guardsize attribute
Summary of function categories

- `pthread_attr_getinheritsched()`: Get the thread inherit scheduling attribute
- `pthread_attr_getschedparam()`: Get the thread scheduling parameters attribute
- `pthread_attr_getschedpolicy()`: Get the thread scheduling policy attribute
- `pthread_attr_getscope()`: Get the thread contention scope attribute
- `pthread_attr_getstackaddr()`: Get the thread stack address attribute
- `pthread_attr_getstacklazy()`: Get thread stack attribute
- `pthread_attr_getstacksize()`: Get the thread stack size attribute
- `pthread_attr_init()`: Initialize thread attribute object
- `pthread_attr_setdetachstate()`: Set the thread detach state attribute
- `pthread_attr_setguardsize()`: Set the thread guardsize attribute
- `pthread_attr_setinheritsched()`: Set the thread inherit scheduling attribute
- `pthread_attr_setschedparam()`: Set the thread scheduling parameters attribute
- `pthread_attr_setschedpolicy()`: Set the thread scheduling policy attribute
Summary of function categories

`pthread_attr_setscope()`
Set the thread contention scope attribute

`pthread_attr_setstackaddr()`
Set the thread stack address attribute

`pthread_attr_setstacklazy()`
Set thread stack attribute

`pthread_attr_setstacksize()`
Set the thread stack size attribute

`pthread_barrierattr_destroy()`
Destroy barrier attributes object

`pthread_barrierattr_getpshared()`
Get process-shared attribute of barrier attributes object

`pthread_barrierattr_init()`
Initialize barrier attributes object

`pthread_barrierattr_setpshared()`
Set process-shared attribute of barrier attributes object

`pthread_barrier_destroy()`
Destroy a barrier object

`pthread_barrier_init()`
Initialize a barrier object

`pthread_barrier_wait()`
Synchronize at a barrier

`pthread_cancel()`
Cancel thread
Summary of function categories

`pthread_cleanup_pop()`
Pop the cancellation cleanup handler

`pthread_cleanup_push()`
Push the cancellation cleanup handler

`pthread_condattr_destroy()`
Destroy the condition variable attribute object

`pthread_condattr_getclock()`
Get the clock selection condition variable attribute

`pthread_condattr_getpshared()`
Get the process-shared attribute from a condition variable attribute object

`pthread_condattr_init()`
Initialize the condition variable attribute object

`pthread_condattr_setclock()`
Set the clock selection condition variable attribute

`pthread_condattr_setpshared()`
Set the process-shared attribute in a condition variable attribute object

`pthread_cond_broadcast()`
Unblock threads waiting on a condition

`pthread_cond_destroy()`
Destroy the condition variable

`pthread_cond_init()`
Initialize the condition variable

`pthread_cond_signal()`
Unblock the thread waiting on condition variable
pthread_cond_timedwait()  
Timed wait on the condition variable

pthread_cond_wait()  
Wait on the condition variable

pthread_create()  
Create a thread

pthread_detach()  
Detach a thread from a process

pthread_equal()  
Compare two thread IDs

pthread_exit()  
Terminate the thread

pthread_getconcurrency()  
Get the level of thread concurrency

pthread_getcpuclockid()  
Return the clock ID of the CPU-time clock from a specified thread

pthread_getschedparam()  
Get the thread scheduling parameters

pthread_getspecific()  
Get the thread specific data value

pthread_join()  
Join the thread

pthread_key_create()  
Create the thread-specific data key

pthread_key_delete()  
Delete the thread-specific data key

pthread_kill()  
Send a signal to a thread
pthread_mutexattr_destroy()
    Destroy the mutex attribute object

pthread_mutexattr_getprioceiling()
    Get the priority ceiling of a mutex attribute object

pthread_mutexattr_getprotocol()
    Get a mutex’s scheduling protocol

pthread_mutexattr_getpshared()
    Get the process-shared attribute from a mutex attribute object

pthread_mutexattr_getrecursive()
    Get the recursive attribute from a mutex attribute object

pthread_mutexattr_gettype()
    Get a mutex type

pthread_mutexattr_init()
    Initialize a mutex attribute object

pthread_mutexattr_setprioceiling()
    Set the priority ceiling of a mutex attribute object

pthread_mutexattr_setprotocol()
    Set a mutex’s scheduling protocol

pthread_mutexattr_setpshared()
    Set the process-shared attribute in a mutex attribute object

pthread_mutexattr_setrecursive()
    Set the recursive attribute in a mutex attribute object

pthread_mutexattr_settype()
    Set a mutex type
Summary of function categories

`pthread_mutex_destroy()`
Destroy a mutex

`pthread_mutex_getprioceiling()`
Get a mutex’s priority ceiling

`pthread_mutex_init()`
Initialize a mutex

`pthread_mutex_lock()`
Lock a mutex

`pthread_mutex_setprioceiling()`
Set a mutex’s priority ceiling

`pthread_mutex_timedlock()`
Lock a mutex

`pthread_mutex_trylock()`
Attempt to lock a mutex

`pthread_mutex_unlock()`
Unlock a mutex

`pthread_once()`
Dynamic package initialization

`pthread_slepon_timedwait()`
Make a thread sleep while waiting

`pthread_timedjoin()`
Join a thread, with a time limit

`pthread_rwlockattr_destroy()`
Destroy a read-write lock attribute object

`pthread_rwlockattr_getpshared()`
Get the process-shared attribute of a read-write lock attribute object
Summary of function categories

- **pthread_rwlockattr_init()**
  Create a read-write lock attribute object

- **pthread_rwlockattr_setpshared()**
  Set the process-shared attribute of a read-write lock attribute object

- **pthread_rwlock_destroy()**
  Destroy a read/write lock

- **pthread_rwlock_init()**
  Initialize a read/write lock

- **pthread_rwlock_rdlock()**
  Acquire a shared read lock on a read/write lock

- **pthread_rwlock_timedrdlock()**
  Lock a read-write lock for writing

- **pthread_rwlock_timedwrlock()**
  Attempt to acquire an exclusive write lock on a read/write lock

- **pthread_rwlock_tryrdlock()**
  Attempt to acquire a shared read lock on a read/write lock

- **pthread_rwlock_trywrlock()**
  Attempt to acquire an exclusive write lock on a read/write lock

- **pthread_rwlock_unlock()**
  Unlock a read/write lock

- **pthread_rwlock_wrlock()**
  Acquire an exclusive write lock on a read/write lock

- **pthread_self()**
  Get the calling thread’s ID
Summary of function categories

pthread_setcancelstate()
Set a thread’s cancellation state

 pthread_setcanceltype()
Set a thread’s cancellation type

pthread_setconcurrency()
Set the concurrency level for a thread

pthread_setschedparam()
Set the thread scheduling parameters

pthread_setspecific()
Set a thread-specific data value

pthread_sigmask()
Examine and change blocked signals

 pthread_sleepon_broadcast()
Unblock waiting threads

pthread_sleepon_lock()
Lock the pthread_sleepon*() functions

pthread_sleepon_signal()
Signal a sleeping thread

pthread_sleepon_unlock()
Unlock the pthread_sleepon*() functions

pthread_sleepon_wait()
Make a thread sleep while waiting

pthread_spin_destroy()
Destroy a thread spinlock

pthread_spin_init()
Initialize a thread spinlock
Summary of functions

Thread functions

`pthread_spin_lock()`
- Lock a thread spinlock

`pthread_spin_trylock()`
- Try to lock a thread spinlock

`pthread_spin_unlock()`
- Unlock a thread spinlock

`pthread_testcancel()`
- Test the thread cancellation

`_sleEMON_broadcast()`
- Wake up multiple threads

`_sleEMON_destroy()`
- Destroy a sleemon lock

`_sleEMON_init()`
- Initialize a sleemon lock

`_sleEMON_lock()`
- Lock a sleemon lock

`_sleEMON_signal()`
- Wake up a single thread

`_sleEMON_unlock()`
- Unlock a sleemon lock

`_sleEMON_wait()`
- Wait on a sleemon lock

Time functions

These functions are concerned with dates and times. (Some of these functions have wide-character versions in the “Wide-character functions” section of the function summary.)

`asctime()`, `asctime_r()`
- Convert time information to a string
Summary of function categories

clock() Return the number of clock ticks used by the program

ClockAdjust(), ClockAdjust_r() Adjust the time of a clock

ClockCycles() Get the number of clock cycles

clock_getcpuclockid() Return the clock ID of the CPU-time clock from a specified process

ClockId(), ClockId_r() Get a clock ID for a given process and thread

ClockPeriod(), ClockPeriod_r() Get or set a clock period

ClockTime(), ClockTime_r() Get or set a clock

ctime(), ctime_r() Convert calendar time to local time

daylight Indicator of support for daylight saving time in the locale

difftime() Calculate the difference between two times

ftime() Get the current time, and store it in a structure
gmtime() Get the current time
gmtime_r() Convert calendar time to a broken-down time

localtime() Convert calendar time to local time

localtime_r() Convert calendar time to local time
Summary of function categories

`mktime()` Convert local time to calendar time

`settimeofday()` Set the time and date

`strftime()` Format a time into a string

`time()` Determine the current calendar time

`TimerAlarm()`, `TimerAlarm_r()` Send an alarm signal

`TimerCreate()`, `TimerCreate_r()` Create a timer for a process

`TimerDestroy()`, `TimerDestroy_r()` Destroy a process timer

`TimerInfo()`, `TimerInfo_r()` Get information about a timer

`TimerSettime()`, `TimerSettime_r()` Set the expiration time for a timer

`timer_timeout()`, `timer_timeout_r()` Set a timeout on a blocking state

`TimerTimeout()`, `TimerTimeout_r()` Set a timeout on a blocking state

`times()` Get time-accounting information

`timezone` The number of seconds by which the local time zone is earlier than UTC

`tm` Structure that describes calendar time

`tzname` The abbreviations for the time zone for standard and daylight savings time

`tzset()` Set the time according to the current time zone
Variable-length argument list functions

Variable-length argument lists are used when a function doesn’t have a fixed number of arguments. These macros provide the capability to access these arguments:

*va_arg()* Get the next item in a list of variable arguments
*va_copy()* Make a copy of a variable argument list
*va_end()* Finish getting items from a variable argument list
*va_start()* Start getting items from a variable argument list

Wide-character functions

If your application must use international characters, you’ll probably need to work with Unicode and wide characters. The functions in this section are wide-character versions of many functions from the following function summary categories:

- Character manipulation functions
- Memory manipulation functions
- Stream I/O functions
- String manipulation functions
- Time functions
- Multibyte character functions
- Searching and sorting functions

The functions are:

*btowc()* Convert a single-byte character to a wide character
*fgetwc()* Read a wide character from a stream
*fgetws()* Read a string of wide characters from a stream
### Summary of Function Categories

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fputwc()</code></td>
<td>Write a wide character to a stream</td>
</tr>
<tr>
<td><code>fputws()</code></td>
<td>Write a wide character string to an output stream</td>
</tr>
<tr>
<td><code>fwide()</code></td>
<td>Set the stream orientation</td>
</tr>
<tr>
<td><code>fwprintf()</code></td>
<td>Write wide-character output to a stream</td>
</tr>
<tr>
<td><code>fwscanf()</code></td>
<td>Scan wide-character input from a stream</td>
</tr>
<tr>
<td><code>getwc()</code></td>
<td>Read a wide character from stdin</td>
</tr>
<tr>
<td><code>getwchar()</code></td>
<td>Read a wide character from a stream</td>
</tr>
<tr>
<td><code>iswalnum()</code></td>
<td>Test for an alphabetic or a decimal digit wide character</td>
</tr>
<tr>
<td><code>iswalpha()</code></td>
<td>Test for an alphabetic wide character</td>
</tr>
<tr>
<td><code>iswcntrl()</code></td>
<td>Test for a control wide character</td>
</tr>
<tr>
<td><code>iswctype()</code></td>
<td>Test for an alphabetic or a decimal digit wide character</td>
</tr>
<tr>
<td><code>iswdigit()</code></td>
<td>Test for a decimal digit wide character</td>
</tr>
<tr>
<td><code>iswgraph()</code></td>
<td>Test for any graphical wide character</td>
</tr>
<tr>
<td><code>iswlower()</code></td>
<td>Test for a lowercase letter wide character</td>
</tr>
<tr>
<td><code>iswprint()</code></td>
<td>Test for a printable wide character</td>
</tr>
<tr>
<td><code>iswpunct()</code></td>
<td>Test for any punctuation wide character</td>
</tr>
<tr>
<td><code>iswspace()</code></td>
<td>Test for a whitespace wide character</td>
</tr>
<tr>
<td><code>iswupper()</code></td>
<td>Test for an uppercase wide character</td>
</tr>
<tr>
<td><code>iswxdigit()</code></td>
<td>Test for any hexadecimal digit wide character</td>
</tr>
<tr>
<td><code>putwc()</code></td>
<td>Write a wide character to a stream</td>
</tr>
<tr>
<td><code>putwchar()</code></td>
<td>Write a wide character to a stdout</td>
</tr>
<tr>
<td><code>swprintf()</code></td>
<td>Print formatted wide-character output into a string</td>
</tr>
</tbody>
</table>
swscanf()  Scan input from a wide character string

towctrans()  Convert a wide character in a specified manner

towlower()  Convert a wide character to lowercase

towupper()  Convert a wide character to uppercase

ungetwc()  Push a wide character back onto an input stream

vfwscanf()  Scan input from a file (varargs)

vfwscanf()  Scan input from a file (varargs)

vswscanf()  Scan input from a string (varargs)

vwscanf()  Scan input from standard input (varargs)

wcrtomb()  Convert a wide-character code into a multibyte character (restartable)

wcscat()  Concatenate two wide-character strings

wcschr()  Find the first occurrence of a wide character in a string

wcscmp()  Compare two wide-character strings

wcscoll()  Compare two wide-character strings, using the locale’s collating sequence

wcscpy()  Copy a wide-character string

wcscspn()  Count the wide characters at the beginning of a string that aren’t in a given character set

wcsftime()  Format the time into a wide-character string
Summary of function categories

- **wcslen()**  Compute the length of a wide-character string
- **wcsncat()**  Concatenate two wide-character strings, up to a maximum length
- **wcsncmp()**  Compare two wide-character strings, up to a given length
- **wcsncpy()**  Copy a wide-character string, to a maximum length
- **wcsrchr()**  Find the last occurrence of a wide character in a string
- **wcspbrk()**  Find the first wide character in a string that’s in a given character set
- **wcsrtombs()**  Convert a wide-character string into a multibyte character string (restartable)
- **wcstod(), wcsf(), wcstold()**  Convert a wide-character string into a double, float, or long double
- **wcstoi(), wcstou()**  Convert a wide-character string into an integer
- **wcstok()**  Break a wide-character string into tokens
- **wcstol(), wcstoll()**  Convert a wide-character string into a long or long long
- **wcsstombs()**  Convert a wide-character string into a multibyte character string
What's in a function description?

Each description contains the following sections:

**Synopsis:**

This section gives the header files that should be included within a source file that references the function or macro. It also shows an appropriate declaration for the function or for a function that could be

As listed:

- `wcstoul()`, `wcstoull()`: Convert a wide-character string into an unsigned long integer or unsigned long long
- `wcsxfrm()`: Transform one wide-character string into another, to a given length
- `wctob()`: Convert a wide character into a single-byte code
- `wctomb()`: Convert a wide character into a multibyte character
- `wctrans()`: Define a wide-character mapping
- `wctype()`: Define a wide-character class
- `wmemchr()`: Locate the first occurrence of a wide character in a buffer
- `wmemcmp()`: Compare wide characters in two buffers
- `wmemcpy()`: Copy wide characters from one buffer to another
- `wmemmove()`: Copy wide characters from one buffer to another
- `wmemset()`: Set wide characters in memory
- `wprintf()`: Write formatted wide-character output to standard output
- `wscanf()`: Scan formatted wide-character input from standard input
substituted for a macro. This declaration isn’t included in your program; only the header file(s) should be included.

When a pointer argument is passed to a function that doesn’t modify the item indicated by that pointer, the argument is shown with `const` before the argument. For example, the following indicates that the array pointed at by `string` isn’t changed:

```c
const char *string
```

**Arguments:**

This section gives a brief description of the arguments to the function.

**Library:**

The section indicates the library that you need to bind with your application in order to use the function.

To link against a library, use the `-l` option to `qcc`, omitting the `lib` prefix and any extension from the library’s name. For example, to link against `libsocket`, specify `-l socket`. For more information, see the Compiling and Debugging chapter of the Neutrino *Programmer’s Guide*.

**Description:**

This section describes the function or macro.

**Returns:**

This section gives the return value (if any) for the function or macro.

**Errors:**

This section describes the special values that the function might assign to the global variable `errno`.
What’s in a function description?

This section doesn’t necessarily list all of the values that the function could set `errno` to.

See also:

This optional section provides a list of related functions or macros as well as pertinent docs to look for more information.

Examples:

This optional section gives one or more examples of the use of the function. The examples are often just code snippets, not complete programs.

Classification:

This section tells where the function or macro is commonly found, which may be helpful when porting code from one environment to another. Here are the classes:

- **ANSI**
  These functions or macros are defined by the ANSI C99 standard.

- **Large-file support**
  These functions support 64-bit offsets.

- **POSIX 1003.1**
  These functions are specified in the document *Information technology — Portable Operating System Interface (IEEE Std 1003.1, 2004 Edition).*

  This standard incorporates the POSIX 1003.2-1992 and 1003.1-1996 standards, the approved drafts (POSIX 1003.1a, POSIX 1003.1d, POSIX 1003.1g and POSIX 1003.1j) and the Standard Unix specification. A joint technical working group — the Austin Common Standards
Revision Group (CSRG) — was formed to merge these standards.

For an up-to-date status of the many POSIX drafts/standards documents, see the PASC (Portable Applications Standards Committee of the IEEE Computer Society) report at http://www.pasc.org/standing/sd11.html.

A classification of “POSIX 1003.1” can be followed by one or more codes that indicate which option or options the functions belong to. The codes include the following:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADV</td>
<td>Advisory Information</td>
</tr>
<tr>
<td>AIO</td>
<td>Asynchronous Input/Output</td>
</tr>
<tr>
<td>BAR</td>
<td>Barriers</td>
</tr>
<tr>
<td>CPT</td>
<td>Process CPU-Time Clocks</td>
</tr>
<tr>
<td>CS</td>
<td>Clock Selection</td>
</tr>
<tr>
<td>FSC</td>
<td>File Synchronization</td>
</tr>
<tr>
<td>MF</td>
<td>Memory Mapped Files</td>
</tr>
<tr>
<td>ML</td>
<td>Process Memory Locking</td>
</tr>
<tr>
<td>MLR</td>
<td>Range Memory Locking</td>
</tr>
<tr>
<td>MPR</td>
<td>Memory Protection</td>
</tr>
<tr>
<td>MSG</td>
<td>Message Passing</td>
</tr>
<tr>
<td>OB</td>
<td>Obsolescent</td>
</tr>
<tr>
<td>PS</td>
<td>Process Scheduling</td>
</tr>
<tr>
<td>RTS</td>
<td>Realtime Signals Extension</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEM</td>
<td>Semaphores</td>
</tr>
<tr>
<td>SHM</td>
<td>Shared Memory Objects</td>
</tr>
<tr>
<td>SIO</td>
<td>Synchronous Input/Output</td>
</tr>
<tr>
<td>SPI</td>
<td>Spin Locks</td>
</tr>
<tr>
<td>TCT</td>
<td>Thread CPU-Time Clocks</td>
</tr>
<tr>
<td>THR</td>
<td>Threads</td>
</tr>
<tr>
<td>TMO</td>
<td>Timeouts</td>
</tr>
<tr>
<td>TMR</td>
<td>Timers</td>
</tr>
<tr>
<td>TPI</td>
<td>Thread Priority Inheritance</td>
</tr>
<tr>
<td>TPP</td>
<td>Thread Priority Protection</td>
</tr>
<tr>
<td>TPS</td>
<td>Thread Execution Scheduling</td>
</tr>
<tr>
<td>TSA</td>
<td>Thread Stack Address Attribute</td>
</tr>
<tr>
<td>TSF</td>
<td>Thread-Safe Functions</td>
</tr>
<tr>
<td>TSH</td>
<td>Thread Process-Shared Synchronization</td>
</tr>
<tr>
<td>TSS</td>
<td>Thread Stack Size Attribute</td>
</tr>
<tr>
<td>TYM</td>
<td>Typed Memory Objects</td>
</tr>
<tr>
<td>XSI</td>
<td>X/Open Systems Interfaces Extension</td>
</tr>
<tr>
<td>XSR</td>
<td>XSI Streams</td>
</tr>
</tbody>
</table>

If two codes are separated by a space, you need to use both options; if the codes are separated by a vertical bar (|), the functionality is supported if you use either option.

For more information, see the *Standard for Information Technology — Portable Operating System Interface: Base Definitions*. 
### QNX 4

These functions or macros are neither ANSI nor POSIX. They perform a function related to the QNX OS version 4. They may be found in other implementations of C for personal computers with the QNX 4 OS. Use these functions with caution if portability is a consideration.

Any QNX 4 functions in the C library are provided *only* to make it easier to port QNX 4 programs. Don’t use these in QNX Neutrino programs.

### QNX Neutrino

These functions or macros are neither ANSI nor POSIX. They perform a function related to the QNX Neutrino OS. They may be found in other implementations of C for personal computers with the QNX OS. Use these functions with caution if portability is a consideration.

### RFC 2292


### SCTP


### SNMP

Simple Network Management Protocol is a network-management protocol whose base document is *RFC 1067*. It’s used to query and modify network device states.

### SOCKS

These functions are part of the SOCKS package consisting of a proxy server, client programs (*rftp* and *rtelnet*), and a library (*libsocks*) for adapting other applications into new client programs. For more information, see the appendix *SOCKS — A Basic Firewall*.

### Unix

These Unix-class functions reside on some Unix systems, but are outside of the POSIX or ANSI standards.
What’s in a function description?

We’ve created the following Unix categories to differentiate:

- **Legacy Unix**: Functions included for backwards compatibility only. New applications shouldn’t use these functions.
- **Unix**: Other Unix functions.

**Function safety:**

This section summarizes whether or not it’s safe to use the C library functions in certain situations:

**Cancellation point**

Indicates whether calling a function may or may not cause the thread to be terminated if a cancellation is pending.

**Interrupt handler**

An interrupt-safe function behaves as documented even if used in an interrupt handler. Functions flagged as interrupt-unsafe shouldn’t be used in interrupt handlers.

**Signal handler**

A signal-safe function behaves as documented even if called from a signal handler *even if the signal interrupts a signal-unsafe function*.

Some of the signal-safe functions modify `errno` on failure. If you use any of these in a signal handler, asynchronous signals may have the side effect of modifying `errno` in an unpredictable way. If any of the code that can be interrupted checks the value of `errno` (this also applies to library calls, so you should assume that most library calls may internally check `errno`), make sure that your signal handler saves `errno` on entry and restores it on exit.
What's in a function description?

All of the above also applies to signal-unsafe functions, with one exception: if a signal handler calls a signal-unsafe function, make sure that signal doesn’t interrupt a signal-unsafe function.

Thread

A thread-safe function behaves as documented even if called in a multi-threaded environment.

Most functions in the QNX Neutrino libraries are thread-safe. Even for those that aren’t, there are still ways to call them safely in a multi-threaded program (e.g. by protecting the calls with a mutex). Such cases are explained in each function’s description.

The “safety” designations documented in this manual are valid for the this release and could change in future versions. Floating-point functions aren’t safe to use in interrupt handlers or signal handlers.

For a summary, see the Summary of Safety Information appendix.
Manifests are used by C/C++ for compile-time changes or inspection. Here are the defined items:

<table>
<thead>
<tr>
<th>Manifest</th>
<th>Header file to include</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_BEGIN_DECLS</td>
<td>sys/platform.h</td>
<td>Denotes start of C code for a C++ compiled program.</td>
</tr>
<tr>
<td>_BIGENDIAN_</td>
<td>sys/platform.h</td>
<td>Code is compiled for a big-endian target.</td>
</tr>
<tr>
<td>_CHAR_SIGNED_</td>
<td>sys/platform.h</td>
<td>Code is compiled with the char type defaulting to \texttt{signed}.</td>
</tr>
<tr>
<td>_CHAR_UNSIGNED_</td>
<td>sys/platform.h</td>
<td>Code is compiled with the char type defaulting to \texttt{unsigned}.</td>
</tr>
<tr>
<td>_END_DECLS</td>
<td>sys/platform.h</td>
<td>Denotes end of C code for a C++ compiled program</td>
</tr>
<tr>
<td>_INT_BITS_</td>
<td>sys/platform.h</td>
<td>The number of bits in the int datatype.</td>
</tr>
<tr>
<td>_LITTLEENDIAN_</td>
<td>sys/platform.h</td>
<td>Code is compiled for a little-endian target.</td>
</tr>
<tr>
<td>_LONG_BITS_</td>
<td>sys/platform.h</td>
<td>The number of bits in the long datatype.</td>
</tr>
<tr>
<td>_NTO_VERSION</td>
<td>sys/neutrino.h</td>
<td>A version number times 100 (e.g. 2.00 is 200).</td>
</tr>
<tr>
<td>_PTR_BITS_</td>
<td>sys/platform.h</td>
<td>The number of bits in a void pointer.</td>
</tr>
<tr>
<td>_OPTIMIZE_</td>
<td>sys/platform.h</td>
<td>Code is compiled for optimization.</td>
</tr>
</tbody>
</table>

\textit{continued…}
<table>
<thead>
<tr>
<th>Manifest</th>
<th>Header file to include</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QNX</strong></td>
<td>N/A</td>
<td>The target is for a QNX operating system (QNX 4 or QNX Neutrino).</td>
</tr>
<tr>
<td><strong>QNXNTO</strong></td>
<td>N/A</td>
<td>The target is the QNX Neutrino operating system.</td>
</tr>
</tbody>
</table>
QNX Neutrino Functions and Macros
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td>abort() to expm1f()</td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td>fabs() to hypotf()</td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to ltrunc()</td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td>main() to outle32()</td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td>pathconf() to ruserok()</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td>sbrk() to system()</td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td>tan() to ynf()</td>
</tr>
</tbody>
</table>
**Synopsis:**

```
#include <stdlib.h>

void abort( void );
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `abort()` function causes abnormal process termination to occur, unless the signal SIGABRT is caught and the signal handler doesn’t return. The status **unsuccessful termination** is returned to the invoking process by means of the function call `raise(SIGABRT)`.

Under QNX Neutrino, the **unsuccessful termination** status value is 6.

**Returns:**

The `abort()` function doesn’t return to its caller.

**Examples:**

```
#include <stdlib.h>

int main( void )
{
    int major_error = 1;

    if( major_error )
        abort();

    /* You’ll never get here. */
    return EXIT_SUCCESS;
}
```
**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

A strictly-conforming POSIX application can’t assume that the `abort()` function is safe to use in a signal handler on other platforms.

**See also:**

`atexit()`, `close()`, `execl()`, `execlp()`, `execvp()`, `execve()`, `execvp()` , `execve()` , `exit()`, `exit()`, `getenv()`, `main()`, `putenv()`, `sigaction()`, `signal()`, `spawn*()` functions, `system()`, `wait()`, `waitpid()`
**abs()**

Return the absolute value of an integer

**Synopsis:**

```c
#include <stdlib.h>

int abs( int j);
```

**Arguments:**

- *j*  
  The number you want the absolute value of.

**Library:**

- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `abs()` function returns the absolute value of the integer argument *j*. If the result can’t be represented as an `int`, a warning occurs.

**Returns:**

The absolute value of its argument.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d %d %d\n", abs (-5), abs (0), abs (5));
    return EXIT_SUCCESS;
}
```

produces the following output:

```
5 0 5
```
abs()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

cabs(), fabs(), labs()
Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

int accept( int s,
            struct sockaddr * addr,
            socklen_t * addrlen );
```

Arguments:

- `s` A socket that’s been created with `socket()`.
- `addr` A result parameter that’s filled in with the address of the connecting entity, as known to the communications layer. The exact format of the `addr` parameter is determined by the domain in which the connection was made.
- `addrlen` A value-result parameter. It should initially contain the amount of space pointed to by `addr`; on return it contains the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `accept()` function:

1. Extracts the first connection request on the queue of pending connections.
2. Creates a new socket with the same properties of `s`, where `s` is a socket that’s been created with `socket()`, bound to an address with `bind()`, and is listening for connections after a `listen()`.
Allocates a new file descriptor for the socket.

If no pending connections are present on the queue, and the socket isn’t marked as nonblocking, `accept()` blocks the caller until a connection is present. If the socket is marked as nonblocking and no pending connections are present on the queue, `accept()` returns an error as described below. The accepted socket may not be used to accept more connections. The original socket remains open.

If you do a `select()` for read on an unconnected socket (on which a `listen()` has been done), the `select()` indicates when a connect request has occurred. In this way, an `accept()` can be made that won’t block. For more information, see `select()`.

For certain protocols that require an explicit confirmation, `accept()` can be thought of as merely dequeuing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

You can obtain user-connection request data without confirming the connection by:

- Issuing a `recvmsg()` call with a `msg_iovlen` of 0 and a nonzero `msg_controllen`
  
Or

- Issuing a `getsockopt()` request.

Similarly, you can provide user-connection rejection information by issuing a `sendmsg()` call with only the control information, or by calling `setsockopt()`.

**Returns:**

A descriptor for the accepted socket, or -1 if an error occurs (`errno` is set).
Errors:

- **EAGAIN**: Insufficient resources to create the new socket.
- **EBADF**: Invalid descriptor $s$.
- **EFAULT**: The $addr$ parameter isn’t in a writable part of the user address space.
- **EOPNOTSUPP**: The referenced socket isn’t a SOCK_STREAM socket.
- **ESRCH**: Can’t find the socket manager ($npm-ttcpip.so$).
- **EWOULDBLOCK**: The socket is marked nonblocking and no connections are present to be accepted.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `bind()`, `close()`, `connect()`, `listen()`, `select()`, `socket()`
Check to see if a file or directory can be accessed

**Synopsis:**

```c
#include <unistd.h>

int access( const char * path,  
            int amode );
```

**Arguments:**

- `path`  The path to the file or directory that you want to access.
- `amode` The access mode you want to check. This must be either:
  - F_OK — test for file existence.
  - or a bitwise ORing of the following access permissions to be checked, as defined in the header `<unistd.h>`:
    - R_OK — test for read permission.
    - W_OK — test for write permission.
    - X_OK — for a directory, test for search permission. Otherwise, test for execute permission.

**Library:**

`libc`

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `access()` function checks to see if the file or directory specified by `path` exists and if it can be accessed with the file access permissions given by `amode`. However, unlike other functions (`open()` for example), it uses the real user ID and real group ID in place of the effective user and group IDs.
access()

**Returns:**

- \(0\) The file or directory exists and can be accessed with the specified mode.
- \(-1\) An error occurred (\textit{errno} is set).

**Errors:**

- **EACCESS** The permissions specified by \textit{amode} are denied, or search permission is denied on a component of the path prefix.
- **EINVAL** An invalid value was specified for \textit{amode}.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG** The length of the \textit{path} string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
- **ENOENT** A component of the path isn’t valid.
- **ENOSYS** The \textit{access()} function isn’t implemented for the filesystem specified in \textit{path}.
- **ENOTDIR** A component of the path prefix isn’t a directory.
- **EROFS** Write access was requested for a file residing on a read-only file system.

**Examples:**

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    if( argc!= 2 ) {
        fprintf( stderr,
            "use: readable <filename>\n"
        );
        return EXIT_FAILURE;
    }
```
if( !access( argv[1], R_OK ) ) {
    printf( "ok to read %s\n", argv[1] );
    return EXIT_SUCCESS;
} else {
    perror( argv[1] );
    return EXIT_FAILURE;
}

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- chmod()
- eaccess
- errno
- fstat()
- open()
- stat()
acos(), acosh()
Compute the arccosine of an angle

Synopsis:
#include <math.h>

double acos( double x );

float acosh( float x );

Arguments:
x  The cosine for which you want to find the angle.

Library:
libm
Use the -l m option to qcc to link against this library.

Description:
These functions compute the arccosine (specified in radians) of x.

Returns:
The arccosine in the range (0, π).

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set errno to 0, call the function, and then check errno again. These functions don’t change errno if no errors occurred.

Examples:
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", acos(.5) );

    return EXIT_SUCCESS;
}
produces the output:

1.047197

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

asin(), atan(), atan2()
Synopsis:

```c
#include <math.h>

double acosh( double x );

float acoshf( float x );
```

Arguments:

- `x` The value for which you want to compute the inverse hyperbolic cosine.

Library:

```
libm
```

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the inverse hyperbolic cosine (specified in radians) of `x`.

Returns:

The inverse hyperbolic cosine of `x` (specified in radians).

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%.15f\n", acosh( 1.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

0.962423695400624
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`asinh()`, `atanh()`, `cosh()`, `errno`
Synopsis:

```c
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    size_t ai_addrlen;
    char * ai_canonname;
    struct sockaddr * ai_addr;
    struct addrinfo * ai_next
};
```

Description:

The `addrinfo` structure describes address information for use with TCP/IP. To get this information, call `getaddrinfo()`; to free a linked list of these structures, call `freeaddrinfo()`.

The `addrinfo` structure includes these members:

- `ai_flags` Flags. Includes AI_PASSIVE, AI_CANONNAME, and AI_NUMERICHOST. For a complete list, see `<netdb.h>`.
- `ai_family` Protocol family. Includes PF_UNSPEC and PF_INET. For a complete list, see `<sys/socket.h>`.
- `ai_socktype` Socket type. Includes SOCK_STREAM and SOCK_DGRAM. For a complete list, see `<sys/socket.h>`.
- `ai_protocol` Protocol. Includes IPPROTO_TCP and IPPROTO_UDP. For a complete list, see `<netinet/in.h>`.
- `ai_addrlen` The length of the `ai_addr` member.
- `ai_canonname` The canonical name for `nodename`. 
ai_addr Binary socket address.

ai_next A pointer to the next `addrinfo` structure in the linked list.

**Classification:**

POSIX 1003.1

**See also:**

`freeaddrinfo()`, `gai_strerror()`, `getaddrinfo()`
aio_cancel()

Cancel an asynchronous I/O operation

Asynchronous I/O operations aren’t currently supported.

Synopsis:

```c
#include <aio.h>

int aio_cancel( int fd,
                struct aiocb *aiocbptr);
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `aio_cancel()` function attempts to cancel one or more asynchronous I/O requests currently outstanding against a file descriptor.

Returns:

-1; `errno` is set.

Errors:

```
ENOSYS  The `aio_cancel()` function isn’t currently supported.
```

Classification:

POSIX 1003.1 AIO

Safety

Cancellation point  No

`continued`
# aio_cancel()

## Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Asynchronous I/O operations aren’t currently supported.

**Synopsis:**
```
#include <aio.h>

int aio_error( const struct aiocb * aiocbptr );
```

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The *aio_error()* function returns the error status associated with the `aiocb` structure referenced by the `aiocbptr` argument. The error status for an asynchronous I/O operation is the `errno` value that’s set by the corresponding `read()`, `write()`, or `fsync()` operation.

**Returns:**
-1; `errno` is set.

**Errors:**
- `ENOSYS` The *aio_error()* function isn’t currently supported.

**Classification:**
POSIX 1003.1 AIO

**Safety**
Cancellation point No

*continued...*
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Asynchronous I/O operations aren’t currently supported.

**Synopsis:**

```c
#include <aio.h>

int aio_fsync( int op,
               struct aiocb * aiocb );
```

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `aio_fsync()` function asynchronously forces all I/O operations associated with the file indicated by the file descriptor to the synchronized I/O completion state.

**Returns:**

-1; `errno` is set.

**Errors:**

- `ENOSYS` The `aio_fsync()` function isn’t currently supported.

**Classification:**

POSIX 1003.1 AIO

**Safety**

Cancellation point   No

continued...
**aio_fsync()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Asynchronous I/O operations aren’t currently supported.

**Synopsis:**
```
#include <aio.h>

int aio_read( struct aiocb * aiocbptr );
```

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `aio_read()` function asynchronously reads from a file.

**Returns:**
-1; `errno` is set.

**Errors:**
```
ENOSYS   The `aio_read()` function isn’t currently supported.
```

**Classification:**
POSIX 1003.1 AIO

**Safety**
- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes

*continued...*
**aio_read()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

© 2007, QNX Software Systems GmbH & Co. KG.
Asynchronous I/O operations aren’t currently supported.

**Synopsis:**

```c
#include <aio.h>

ssize_t aio_return( struct aiocb * aiocbptr );
```

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `aio_return()` function returns the return status associated with the `aiocb` structure referenced by the `aiocbptr` argument. The return status for an asynchronous I/O operation is the value that’s returned by the corresponding `read()`, `write()`, or `fsync()` operation.

**Returns:**

-1; `errno` is set.

**Errors:**

- `ENOSYS` The `aio_return()` function function isn’t currently supported.

**Classification:**

POSIX 1003.1 AIO
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Asynchronous I/O operations aren’t currently supported.

Synopsis:

```c
#include <aio.h>

int aio_suspend( const struct aiocb * const list[],
                int nent,
                const struct timespec * timeout );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `aio_suspend()` function suspends the calling thread until at least one of the asynchronous I/O operations referenced by the `list` argument has completed, until a signal interrupts the function, or, if `timeout` isn’t NULL, until the time interval specified by `timeout` has passed.

Returns:

-1; `errno` is set.

Errors:

`ENOSYS` The `aio_suspend()` function isn’t currently supported.

Classification:

POSIX 1003.1 AIO
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Asynchronously write to a file

Asynchronous I/O operations aren’t currently supported.

Synopsis:

```c
#include <aio.h>

int aio_write( struct aiocb * aiocbptr );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `aio_write()` function asynchronously writes to a file.

Returns:

-1; `errno` is set.

Errors:

`ENOSYS` The `aio_write()` function isn’t currently supported.

Classification:

POSIX 1003.1 AIO

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
aio_write()
alarm()  
Schedule an alarm

Synopsis:

```c
#include <unistd.h>

unsigned int alarm( unsigned int seconds );
```

Arguments:

`seconds`  
The number of seconds of realtime to let elapse before raising the alarm, or zero to cancel any previous `alarm()` requests.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `alarm()` function causes the system to send the calling process a SIGALRM signal after a specified number of realtime seconds have elapsed. To add a handler for the signal, call `signal()` or `SignalAction()`.

Processor scheduling delays may cause the process to handle the signal after the desired time.

The `alarm()` requests aren’t stacked; you can schedule only a single SIGALRM generation in this manner. If the SIGALRM hasn’t yet been generated, `alarm()` reschedules the time at which the SIGALRM is generated.

Returns:

The number of seconds before the calling process is scheduled to receive a SIGALRM from the system, or zero if there was no previous `alarm()` request.

If an error occurs, an `(unsigned)-1` is returned (`errno` is set).
Errors:

EAGAIN  All timers are in use. You’ll have to wait for a process to release one.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    unsigned int timeleft;

    printf( "Set the alarm and sleep\n" );
    alarm( 10 );
    sleep( 5 );  /* go to sleep for 5 seconds */

    /*
     * To get the time left before the SIGALRM is
     * to arrive, one must cancel the initial timer,
     * which returns the amount of time it had
     * remaining.
     */
    timeleft = alarm( 0 );
    printf( "Time left before cancel, and rearm: %d\n",
            timeleft );

    /*
     * Start a new timer that kicks us when timeleft
     * seconds have passed.
     */
    alarm( timeleft );

    /*
     * Wait until we receive the SIGALRM signal; any
     * signal kills us, though, since we don’t have
     * a signal handler.
     */
    printf( "Hanging around, waiting to die\n" );
    pause();
    return EXIT_SUCCESS;
}
```
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Requests from `alarm()`, `TimerAlarm()`, and `ualarm()` aren’t “stacked;” only a single SIGALRM generator can be scheduled with these functions. If the SIGALRM signal hasn’t been generated, the next call to `alarm()`, `TimerAlarm()`, or `ualarm()` reschedules it.

See also:

`errno`, `pause()`, `signal()`, `SignalAction()`, `sleep()`, `TimerAlarm()`, `timer_create()`, `timer_delete()`, `timer_gettime()`, `timer_settime()`, `ualarm()`
**alloca()**

Allocate automatic space from the stack

**Synopsis:**
```c
#include <alloca.h>

void* alloca( size_t size );
```

**Arguments:**
- **size**  The number of bytes of memory to allocate.

**Library:**
- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `alloca()` function allocates space for an object of `size` bytes from the stack. The allocated space is automatically discarded when the current function exits.

---

Don’t use this function in an expression that’s an argument to a function.

**Returns:**
A pointer to the start of the allocated memory, or NULL if an error occurred (`errno` is set).

**Examples:**
```c
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

FILE *open_err_file( char *name )
{
    char *buffer;

    /* allocate temporary buffer for file name */
```
alloca()

```c
buffer = (char *)alloca(strlen(name) + 5);

if (buffer) {
    FILE *fp;
    sprintf(buffer, "%s.err", name);
    fp = fopen(buffer, "w");
    return fp;
}
return (FILE *)NULL;
}

int main(void)
{
    FILE *fp;
    fp = open_err_file("alloca_test");
    if (fp == NULL)
        printf("Unable to open error file\n");
    else
        fprintf(fp, "Hello from the alloca test.\n");
    fclose(fp);
    return EXIT_SUCCESS;
}
```

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

*Don't use `alloca()` as an argument to a function.*

See also:

`calloc()`, `malloc()`
Synopsis:

```
#include <sys/types.h>
#include <sys/dir.h>

struct direct {
    unsigned long d_fileno;
    unsigned short d_reclen;
    unsigned short d_namlen;
    char d_name[1];
};

int alphasort( struct direct **d1,
               struct direct **d2);
```

Arguments:

- `d1`, `d2` Pointers to the directory entries that you want to compare.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `alphasort()` function alphabetically compares two directory entries. You can use it as the `compar` argument to `scandir()`.

Returns:

- `< 0` The `d1` entry precedes the `d2` entry alphabetically.
- `0` The entries are equivalent.
- `> 0` The `d1` entry follows the `d2` entry alphabetically.
Classification:

Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

closedir(), opendir(), readdir(), rewwnddir(), scandir(), seekdir(), telldir()
Synopsis:

#include <stdlib.h>

unsigned int _amblksiz

Description:

The _amblksiz global variable holds the increment by which the “break” pointer for memory allocation is advanced when there’s no freed block large enough to satisfy a request to allocate a block of memory. You can change this at any time.

This variable represents the unit size that will be used to ask for memory from the system when a core allocation needs to be made. Core allocations are made from the system by using the mmap() call. In the current implementation of the allocator, requests for memory larger than 32 KB are automatically serviced by calling mmap() directly, while smaller allocations are serviced by a split-coalesce mechanism inside the allocator.

Setting this value affects all allocations that are smaller than 32 KB and require a core allocation. Memory that has become free will eventually return to the system when all memory associated with a specific core allocation has been released back to the allocator. Even when a block has been fully released to the allocator, it’s possible for the allocator, for efficiency purposes, to retain some blocks locally within the heap (without releasing memory to the system immediately). This is done to avoid thrashing behavior, when requests to allocate and free memory cause the the allocator to constantly request and release memory to and from the system.

Classification:

QNX Neutrino

See also:

malloc()
The Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino *Programmer’s Guide*. 
The number of arguments passed to main()

Synopsis:

    int _argc

Description:

This global variable holds the number of arguments passed to main().

This variable isn’t defined in any header file. If you want to refer to it, you need to add your own extern statement.

Classification:

QNX Neutrino

See also:

_arvg, _auxv, getopt(), main()
Synopsis:

```c
char ** _argv;
```

Description:

This global variable holds a pointer to a vector containing the actual arguments passed to `main()`.

This variable isn’t defined in any header file. If you want to refer to it, you need to add your own `extern` statement.

Classification:

QNX Neutrino

See also:

`_argc`, `_auxv`, `getopt()`, `main()`
asctime(), asctime_r()

Convert time information to a string

Synopsis:

```c
#include <time.h>

char* asctime( const struct tm* timeptr );

char* asctime_r( const struct tm* timeptr, char* buf );
```

Arguments:

- `timeptr` A pointer to a `tm` structure that contains the time that you want to convert to a string.
- `buf` (asctime_r() only) A buffer in which asctime_r() can store the resulting string. This buffer must be large enough to hold at least 26 characters.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `asctime()` and `asctime_r()` functions convert the time information in the structure pointed to by `timeptr` into a string containing exactly 26 characters, in the form:

```
Tue May  7 10:40:27 2002
```

September 10, 2007 QNX Neutrino Functions and Macros 161
The `asctime()` function places the result string in a static buffer that’s reused every time you call `asctime()` or `ctime()`. Calling `gmtime()` or `localtime()` could also change the date in this static buffer.

The result string for `asctime_r()` is contained in the buffer pointed to by `buf`.

All fields have a constant width. The newline character (`\n`) and a NUL character (`\0`) occupy the last two positions of the string.

**Returns:**

A pointer to the character string result, or NULL if an error occurred.

**Classification:**

`asctime()` is ANSI, POSIX 1003.1; `asctime_r()` is POSIX 1003.1 TSF

### asctime()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

### asctime_r()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

*continued...*
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The `asctime()` and `ctime()` functions place their results in a static buffer that's reused for each call to `asctime()` or `ctime()`.

See also:

`clock()`, `ctime()`, `difftime()`, `gmtime()`, `localtime()`, `localtime_r()`, `mktime()`, `strftime()`, `time()`, `tm`, `tzset()`
Compute the arcsine of an angle

**Synopsis:**
```
#include <math.h>

double asin( double x );

float asinf( float x );
```

**Arguments:**

- `x` The sine for which you want to find the angle.

**Library:**

```
libm
```
Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the value of the arcsine (specified in radians) of `x`.

**Returns:**

The arcsine, in the range (-π/2, π/2).

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", asin(.5) );
    return EXIT_SUCCESS;
}
```
produces the output:

0.523599

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

acos(), atan(), atan2(), errno
Compute the inverse hyperbolic sine

**Synopsis:**

```c
#include <math.h>

double asinh( double x );

float asinhf( float x );
```

**Arguments:**

x  The value for which you want to compute the inverse hyperbolic sine.

**Library:**

libm

Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the inverse hyperbolic sine of x.

**Returns:**

The inverse hyperbolic sine (specified in radians) of x.

**Examples:**

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", asinh( 0.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
0.481212
```
asinh(), asinhf()

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

acosh(), atanh(), sinh(), errno
**assert()**

Print a diagnostic message and optionally terminate the program

**Synopsis:**

```c
#include <assert.h>

void assert( int expression );
```

**Arguments:**

`expression`  
Zero if you want to terminate the program; a nonzero value if you don’t.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `assert()` macro prints a diagnostic message on the `stderr` stream, and terminates the program, using `abort()`, if `expression` is false (0).

The diagnostic message includes the `expression`, the name of the source file (the value of `__FILE__`) and the line number of the failed assertion (the value of `__LINE__`).

No action is taken if `expression` is true (nonzero).

You typically use the `assert()` macro while developing a program, to identify program logic errors. You should choose the `expression` so that it’s true when the program is functioning as intended.

After the program has been debugged, you can use the special “no debug” identifier, NDEBUG, to remove calls to `assert()` from the program when it’s recompiled. If you use the `-D` option to `qcc` or a `#define` directive to define NDEBUG (with any value), the C preprocessor ignores all `assert()` calls in the program source.
To remove the calls to `assert()`, you must define NDEBUG in the code before including the `<assert.h>` header file (i.e. `#include <assert.h>`).

If you define NDEBUG, the preprocessor also ignores the expression you pass to `assert()`. For example, if your code includes:

```c
assert((fd = open("filename", O_RDWR)) != -1);
```

and you define NDEBUG, the preprocessor ignores the entire call to `assert()`, including the call to `open()`.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void process_string( char *string )
{
    /* use assert to check argument */
    assert( string != NULL );
    assert( *string != '\'0' );
    /* rest of code follows here */
}

int main( void )
{
    process_string( "hello" );
    process_string( "" );

    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

Cancellation point No

continued...
assert() is a macro.

See also:

abort(), stderr
asyncmsg_channel_create()
Create an asynchronous message channel

Synopsis:

```c
#include <errno.h>
#include <stdlib.h>
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_channel_create;
    (unsigned flags, mode_t mode, size_t buffer_size,
     unsigned max_num_buffer, const struct sigevent *ev

int (*recvbuf_callback) (size_t bufsize, unsigned num_bufs,
     void *bufs[], int flags));
```

Arguments:

flags Flags that specify the type and attributes of the channel:

- _NTO_CHF_COID_DISCONNECT
- _NTO_CHF_DISCONNECT
- _NTO_CHF_FIXED_PRIORITY
- _NTO_CHF_NET_MSG
- _NTO_CHF_REPLY_LEN
- _NTO_CHF_SENDER_LEN
- _NTO_CHF_THREAD_DEATH
- _NTO_CHF_UNBLOCK

mode Access the permission of the channel. This is the same as the permission of a file.

buffer_size The size of each buffer used to store messages.

max_num_buffer The maximum number of buffers to be allocated. A size of 0 means there’s no buffer.
asyncmsg_channel_create() © 2007, QNX Software Systems GmbH & Co. KG.

- **ev**: The event to be sent for notification. When the `ev` argument is not NULL, a sigevent `ev` delivers a message when available, for a queue that was previously empty.

- **recvbuf_callback**: A callback function used by the library to allocate the buffer for incoming messages, or free buffers, if the channel is destroyed. If the callback is NULL, the library will use `malloc()` and `free()` instead.

**Library:**

- **libasyncmsg**

  Use the `-l asyncmsg` option to qcc to link against this library.

**Description:**

The `asyncmsg_channel_create()` function creates an asynchronous message channel.

**Returns:**

The channel ID of the newly created channel, or -1 if an error has occurred (`errno` is set).

**Errors:**

- **EAGAIN**: All kernel channel objects are in use.
- **EINVAL**: Attach to a synchronous message channel.

**Classification:**

- QNX Neutrino
asyncmsg_channel_create()

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asyncmsg_channel_destroy(), asyncmsg_connect_attach(),
asyncmsg_connect_attr(), asyncmsg_connect_detach(),
asyncmsg_flush(), asyncmsg_free(), asyncmsg_get(),
asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

ChannelCreate()

Asynchronous Messaging Technote
asyncmsg_channel_destroy() © 2007, QNX Software Systems GmbH & Co.

Destroy an asynchronous message channel

Synopsis:

```c
#include <stdlib.h>
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_channel_destroy(int chid)
```

Arguments:

- `chid` The channel ID to be destroyed.

Library:

- `libasyncmsg`
  Use the `-l asyncmsg` option to `qcc` to link against this library.

Description:

The `asyncmsg_channel_destroy()` function destroys the asynchronous message channel specified by `chid`.

Returns:

- EOK for success; or -1 if an error occurred (`errno` is set).

Errors:

- `EINVAL` The channel specified by `chid` doesn’t exist.

Classification:

- QNX Neutrino

  Safety

  - Cancellation point: No
  - Interrupt handler: No

  continued...
asyncmsg_channel_destroy()

Safety

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asyncmsg_channel_create(), asyncmsg_connect_attach(),
asyncmsg_connect_attr(), asyncmsg_connect_detach(),
asyncmsg_flush(), asyncmsg_free(), asyncmsg_get(),
asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

ChannelDestroy()

Asynchronous Messaging Technote
asyncmsg_connectAttach() © 2007, QNX Software Systems GmbH & Co. KG.

Establish a connection between a process and a channel

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_connectAttach;
    (uint32_t nd, pid_t pid, int chid,
     unsigned index, unsigned flags,
     const struct _asyncmsg_connection_attr *attr);
```

Arguments:

- `nd` The node descriptor.
- `pid` The process ID of the owner of the channel.
- `chid` The channel ID, returned by asyncmsg_channel_create(), of the channel to connect to the process.
- `index` The value of the connection ID.
- `flags` Flags that specify the type and attributes of the connection:
  - `NTO_COF_NOSHARE` User wants to use its own buffer; otherwise the user gets the buffer from the asyncmsg_malloc() call, fills it in and sends it by calling the asyncmsg_put() function.
  - `NTO_COF_NONBLOCK` Don’t block waiting if all the send headers are in use.
- `attr` The connection attributes. If call_back is not NULL, this function will be called when an error occurs during send (after asyncmsg_put() returns) with an error number in err and the faulted buffer in buff. If the user uses its own buffer, this function will also be called when a buffer is empty, with err being EOK.
asyncmsg_connect_attach()

Library:

`libasyncmsg`
Use the `-l asyncmsg` option to `qcc` to link against this library.

Description:

The `asyncmsg_connect_attach()` function establishes a connection between the calling process and a channel identified by the arguments `nd`, `pid` and `chid`. The system returns the first available connection ID starting at the value specified by the `index` argument.

Returns:

A connection ID on success; or -1 if an error occurred (`errno` is set).

Errors:

EAGAIN    All kernel channel objects are in use.
ESRCH     The node indicated by `nd`, the process indicated by `pid` or the channel indicated by `chid` does not exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

asyncmsg_channel_create(), asyncmsg_channel_destroy(),
asyncmsg_connect_attr(), asyncmsg_connect_detach(),
asyncmsg_flush(), asyncmsg_free(), asyncmsg_get(),
asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

ConnectAttach()
**Synopsis:**

```c
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_connect_attr;
    (int coid, struct _asyncmsg_connection_attr *old_attr,
     const struct _asyncmsg_connection_attr *new_attr);
```

**Arguments:**

- `coid` The connection ID.
- `old_attr` The attributes for the original connection.
- `new_attr` The attributes for the new connection.

**Library:**

`libasyncmsg`

Use the `-l asyncmsg` option to `qcc` to link against this library.

**Description:**

The `asyncmsg_connect_attr()` function returns the original connection attributes in the buffer pointed to by `old_attr` if `old_attr` is not NULL and sets the connection attributes to the contents of the buffer pointed to by `new_attr` if `new_attr` is not NULL.

**Returns:**

- EOK on success; or -1 if an error occurred (`errno` is set).

**Errors:**

- EINVAL The connection specified by `coid` doesn’t exist.
**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

asyncmsg_channel_create(), asyncmsg_channel_destroy(), asyncmsg_connect_attach(), asyncmsg_connect_detach(), asyncmsg_flush(), asyncmsg_free(), asyncmsg_get(), asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

Asynchronous Messaging Technote
**Synopsis:**

```c
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_connect_detach(int coid)
```

**Arguments:**

- `coid` The connection ID of the connection you want to break.

**Library:**

`libasyncmsg`

Use the `-l asyncmsg` option to `qcc` to link against this library.

**Description:**

The `asyncmsg_connect_detach()` function breaks the connection specified by the connection ID `coid` argument. All the messages buffered on the sender side will be discarded. If the user wants to ensure that all the messages sent have been delivered, call `asyncmsg_flush()` before calling this function.

**Returns:**

EOK on success; or -1 if an error occurred (errno is set).

**Errors:**

- `EINVAL` The connection specified by `coid` doesn’t exist.

**Classification:**

QNX Neutrino
**asyncmsg_connect_detach()**

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

- `asyncmsg_channel_create()`, `asyncmsg_channel_destroy()`,
- `asyncmsg_connect_attach()`, `asyncmsg_connect_attr()`,
- `asyncmsg_flush()`, `asyncmsg_free()`, `asyncmsg_get()`,
- `asyncmsg_malloc()`, `asyncmsg_put()`, `asyncmsg_putv()`

*ConnectDetach()*

*Asynchronous Messaging* Technote
asyncmsg_connection_attr

Defines the connection attributes used to receive an asynchronous message.

Synopsis:

```c
struct _asyncmsg_connection_attr {
    int (*call_back)(int err, void* buff, unsigned handle);
    size_t buffer_size;
    unsigned max_num_buffer;
    unsigned trigger_num_msg;
    struct itimer trigger_timer;
};
```

Description:

The `_asyncmsg_connection_attr` structure describes connection attributes for use with asynchronous messaging.

The `_asyncmsg_connection_attr` structure includes these members:

- `*call_back`  Callback function for notification.
- `err`         The error status of the package.
- `buff`        A pointer to the package buffer.
- `handle`      A handle associated with the callback function.
- `buffer_size` The message buffer size.
- `max_num_buffer`  The maximum number of buffers allowed for this connection.
- `trigger_num_msg`  Uses the number of the pending message as triggering criteria.
- `trigger_timer`  Uses the time passed since the last kernel call as triggering criteria.
Classification:

QNX Neutrino

See also:

asyncmsg_connect_attach(), asyncmsg_connect_attr()

Asynchronous Messaging Technote
asyncmsg_flush()

Flush the messages sent through the connection

Synopsis:

```
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_flush (int coid)
```

Arguments:

- `coid` The connection ID of the connection you want to flush.

Library:

```
libasyncmsg
```

Use the `-l asyncmsg` option to `qcc` to link against this library.

Description:

The `asyncmsg_flush()` function flushes the messages sent through the connection specified by the connection ID `coid` argument.

The function will not return until all the existing messages are delivered to the receive side.

Returns:

- EOK on success; or -1 if an error occurred (`errno` is set).

Errors:

- EBADF The connection specified by `coid` doesn’t exist.

Classification:

```
QNX Neutrino
```

Safety

```
Cancellation point  No
```

continued…
asyncmsg_flush()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asyncmsg_channel_create(), asyncmsg_channel_destroy(), asyncmsg_connect_attach(), asyncmsg_connect_attr(), asyncmsg_connect_detach(), asyncmsg_free(), asyncmsg_get(), asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

Asynchronous Messaging Technote
**asyncmsg_free()**

Free a message buffer

**Synopsis:**

```c
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

void asyncmsg_free(void *buf)
```

**Library:**

```c
libasyncmsg
```

Use the `-l asyncmsg` option to `qcc` to link against this library.

**Description:**

The `asyncmsg_free()` function frees a message buffer that comes from the `asyncmsg_get()` call.

**Returns:**

None

**Errors:**

None

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

asyncmsg_channel_create(), asyncmsg_channel_destroy(),
asyncmsg_connect_attach(), asyncmsg_connect_attr(),
asyncmsg_connect_detach(), asyncmsg_flush(), asyncmsg_get(),
asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

Asynchronous Messaging Technote
**asyncmsg_get()**

*Receive an asynchronous message*

**Synopsis:**

```c
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

struct asyncmsg_get_header* asyncmsg_get(int chid)
```

**Arguments:**

- `chid` The channel ID.

**Library:**

`libasyncmsg`

Use the `-l asyncmsg` option to `qcc` to link against this library.

**Description:**

The `asyncmsg_get()` function receives one to five asynchronous messages from the channel identified by the `chid` argument. In order to receive more messages, you must call this function in a loop until you get a NULL return and EAGAIN to signify that you’ve drained the queue of messages.

The above description is the current behavior that may change in future.

**Returns:**

A pointer to `_asyncmsg_get_header()`, if the message was successfully received; or NULL if an error occurred (`errno` is set). The `_asyncmsg_get_header()`, call provides the receive message header used to receive an asynchronous message.
asyncmsg_get()

Errors:

- **EBADF** The channel specified by `chid` doesn’t exist.
- **EFAULT** A fault occurred when the kernel tried to access the buffers provided.
- **EMSGSIZE** The buffer provided isn’t big enough to hold the received message.
- **EAGAIN** No message is available at this time.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

asyncmsg_channel_create(), asyncmsg_channel_destroy(),
asyncmsg_connect_attach(), asyncmsg_connect_attr(),
asyncmsg_connect_detach(), asyncmsg_flush(), asyncmsg_free(),
asyncmsg_malloc(), asyncmsg_put(), asyncmsg_putv()

Asyncronous Messaging Technote
asyncmsg_malloc()
Allocate a message buffer for sending

Synopsis:
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

void *asyncmsg_malloc(size_t size)

Arguments:

size The size of the message.

Library:

libasyncmsg
Use the -l asyncmsg option to qcc to link against this library.

Description:

The asyncmsg_malloc() function allocates a message buffer for sending.

Returns:

EOK on success; or -1 if an error occurred (errno is set).

Errors:

ENOMEM There's not enough memory.

Classification:

QNX Neutrino

Safety

Cancellation point No
Interrupt handler No

continued . . .
asyncmsg_malloc()

Safety

<table>
<thead>
<tr>
<th>Signal handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asyncmsg_channel_create(), asyncmsg_channel_destroy(),
asyncmsg_connect_attach(), asyncmsg_connect_attr(),
asyncmsg_connect_detach(), asyncmsg_flush(), asyncmsg_free(),
asyncmsg_get(), asyncmsg_put(), asyncmsg_putv()

Asynchronous Messaging Technote
asyncmsg_put(),
asyncmsg_putv()

Send an asynchronous message to a connection

Synopsis:

```c
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_put(int coid, const void *buff, size_t size, 
                 unsigned handler),

int asyncmsg_putv(int coid, const iov_t* iov, int parts, 
                   unsigned handler,

int (*call_back) (int err, void* buf, unsigned handler)
```

Arguments:

coid The connection ID the message is sent to.

buff A pointer to the buffer where the message comes from.

size The size of the message.

iov A pointer to an array of buffers where the message is taken from.

parts The number of elements in the array.

handler A user-defined handle that’s passed back in the call_back function to allow for quick identification of the message’s package.

call_back The call_back function tells the application that one of the messages is processed. If NULL is returned the call_back in the asyncmsg_connectAttach() will be used.
asyncmsg_put(), asyncmsg_putv() © 2007, QNX Software Systems GmbH & Co. KG.

Library:

libasyncmsg

Use the -l asyncmsg option to qcc to link against this library.

Description:

The asyncmsg_put() and asyncmsg_putv() functions take asynchronous messages from the buffer pointed to by buff, and sends them to the connections identified by the coid arguments. The messages could be associated with a user-defined handle which will be passed back in the call_back function (for error or buffer claim notification) to allow the user to quickly find out which package it is.

The call_back function tells the application that one of the messages is processed. If NULL is returned the call_back in the asyncmsg_connect_attach() will be used.

Returns:

EOK on success; or -1 if an error occurred (errno is set).

Errors:

EBADF The connection specified by coid doesn’t exist.
EFault A fault occurred when the kernel tried to access the buffers provided.
EAGAIN The send queue is full.

Classification:

QNX Neutrino

Safety

Cancellation point  No

continued . . .
**asyncmsg_put(), asyncmsg_putv()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

asyncmsg_channel_create(), asyncmsg_channel_destroy(),
asyncmsg_connect_attach(), asyncmsg_connect_attr(),
asyncmsg_connect_detach(), asyncmsg_flush(), asyncmsg_free(),
asyncmsg_get(), asyncmsg_malloc()

*Asynchronous Messaging* Technote
**atan(), atanf()**

Compute the arctangent of an angle

**Synopsis:**

```c
#include <math.h>

double atan( double x );

float atanf( float x );
```

**Arguments:**

- `x` The tangent for which you want to find the angle.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the arctangent (specified in radians) of `x`.

**Returns:**

The arctangent, in the range `(-\pi/2, \pi/2)`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", atan(.5) );
    return EXIT_SUCCESS;
}
```
produces the output:

\[ 0.463648 \]

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`acos()`, `asin()`, `atan2()`
**atan2(), atan2f()**

Compute the arctangent, determining the quadrant

**Synopsis:**

```c
#include <math.h>

double atan2( double y, double x );

float atan2f( float y, float x );
```

**Arguments:**

- `x, y` The value \( y/x \) for which you want to find the angle.

**Library:**

```
libm
```

Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the value of the arctangent (specified in radians) of \( y/x \), using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero.

**Returns:**

The arctangent of \( y/x \), in the range \((-\pi, \pi)\).

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
atan2(), atan2f()

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", atan2( .5, 1. ) );
    return EXIT_SUCCESS;
}
```

produces the output:

0.463648

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`acos()`, `asin()`, `atan()`, `errno`
**atanh(), atanhf()**

Compute an inverse hyperbolic tangent

**Synopsis:**
```
#include <math.h>

double atanh( double x );
float atanhf( float x );
```

**Arguments:**

- `x`  
The value for which you want to compute the inverse hyperbolic tangent.

**Library:**
```
libm
```

Use the `-l m` option to `qcc` to link against this library.

**Description:**
These functions compute the inverse hyperbolic tangent (specified in radians) of `x`.

**Returns:**
The inverse hyperbolic tangent of `x`.

**Examples:**
```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", atanh( 0.5 ) );
    return EXIT_SUCCESS;
}
```
produces the output:
```
0.549306
```
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`acosh(), asinh(), errno, tanh()`
atexit()

Register functions to be called during normal program termination

Synopsis:
#include <stdlib.h>

    int atexit( register void (*func)(void) );

Arguments:

    func  A pointer to the function you want to be called when the program terminates normally. This function has no arguments and doesn’t return a value; its prototype should be:

            void func( void );

Library:

    libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

    The atexit() function registers a function to be called when the program terminates normally. If you register more than one function with atexit(), they’re executed in a “last-in, first-out” order. Normal termination occurs either by a call to exit() or a return from main().

You can register a total of 32 functions with atexit().

The functions registered with atexit() aren’t called when the program terminates with a call to _exit().

Returns:

    0 for success, or nonzero if an error occurs.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

void func1( void )
{
    printf( "last.\n" );
}

void func2( void )
{
    printf( "this " );
}

void func3( void )
{
    printf( "Do " );
}

int main( void )
{
    atexit( func1 );
    atexit( func2 );
    atexit( func3 );
    printf( "Do this first.\n" );
    return EXIT_SUCCESS;
}
```

produces the output:

```
Do this first.
Do this last.
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
atexit()

Safety
---
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abort(), _exit(), exit()
Synopsis:

```c
#include <stdlib.h>

double atof( const char* ptr );
```

Arguments:

`ptr`  
A pointer to the string to parse.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atof()` function converts the string pointed to by `ptr` to a `double`. Calling it is equivalent to calling `strtod()` like this:

```c
strtod( ptr, (char**)NULL )
```

Returns:

The converted `double`, or `0.0` if an error occurs.

Errors:

If an error occurs, `errno` is set to `ERANGE`.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    double x;

    x = atof( "3.1415926" );
    printf( "%x = %f
", x );
    return EXIT_SUCCESS;
}
```
atof()

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, sscanf(), strtol()
Synopsis:

```c
#include <stdlib.h>

unsigned atoh( const char* ptr );
```

Arguments:

- **ptr** A pointer to the string to parse.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atoh()` function converts the string pointed to by `ptr` to `unsigned` representation, assuming the string contains a hexadecimal (base 16) number.

Returns:

The converted value.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    unsigned x;

    x = atoh( "F1A6" );
    printf( "number is %x\n", x );
    return EXIT_SUCCESS;
}
```
**atoh()**  

© 2007, QNX Software Systems GmbH & Co. KG.

---

**Classification:**

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`sscanf()`
Synopsis:
#include <stdlib.h>

    int atoi( const char* ptr );

Arguments:

    ptr   A pointer to the string to parse.

Library:

    libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The atoi() function converts the string pointed to by ptr to an int.

Returns:

The converted integer.

Examples:
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int x;

    x = atoi( "-289" );
    printf( "%d\n", x );
    return EXIT_SUCCESS;
}

produces the output:

    x = -289
**atoi()**

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atol()`, `itoa()`, `ltoa()`, `sscanf()`, `strtol()`, `strtol()`, `ultoa()`, `utoa()`
atol(), atoll()
Convert a string into a long integer

Synopsis:

```c
#include <stdlib.h>

long atol( const char* ptr );
long long atoll( const char* ptr );
```

Arguments:

`ptr` A pointer to the string to parse.

Library:

`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atol()` function converts the string pointed to by `ptr` to a `long` integer; `atoll()` converts the string pointed to by `nptr` to a `long long` integer.

Returns:

`atol()` A `long` integer.

`atoll()` A `long long` integer.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    long x;
    x = atol( "-289" );
    printf( "%d\n", x );
    return EXIT_SUCCESS;
}
```
produces the output:
\[ x = -289 \]

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atol()`, `itoa()`, `ltoa()`, `sscanf()`, `strtol()`, `strtoul()`, `ultoa()`, `utoa()`
Synopsis:

```c
#include <atomic.h>

void atomic_add( volatile unsigned * loc,
                 unsigned incr );
```

Arguments:

- `loc` A pointer to the value that you want to add to.
- `incr` The number that you want to add.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_add()` function is a thread-safe way of doing an `(*loc) += incr` operation, even in a symmetric-multiprocessing system.

The `atomic_*()` functions are guaranteed to complete without being preempted by another thread.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*()` functions.

The `atomic_*()` functions are useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

Examples:

To safely increment a counter shared between multiple threads:

```c
#include <atomic.h>
...
```
volatile unsigned count;
...

atomic_add( &count, 1 );

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

atomic_add_value(), atomic_clr(), atomic_clr_value(), atomic_set(), atomic_set_value(), atomic_sub(), atomic_sub_value(), atomic_toggle(), atomic_toggle_value()
atomic_add_value()
Safely add to a variable, returning the previous value

Synopsis:
#include <atomic.h>

unsigned atomic_add_value( volatile unsigned * loc,
                           unsigned incr );

Arguments:
loc A pointer to the value that you want to add to.
incr The number that you want to add.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The atomic_add_value() function is a thread-safe way of doing an (*loc) += incr operation, even in a symmetric-multiprocessing system.

The atomic_*() functions are guaranteed to complete without being preempted by another thread.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the atomic_*() functions.

The atomic_*() functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

The atomic_add_value() function may be slower than atomic_add().
atomic_add_value()

Returns:

The previous value of loc’s contents.

Examples:

To safely increment a counter shared between multiple threads:

```c
#include <atomic.h>
...
volatile unsigned count;
unsigned previous;
...

previous = atomic_add_value( &count, 1 );
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

atomic_add(), atomic_clr(), atomic_clr_value(), atomic_set(),
atomic_set_value(), atomic_sub(), atomic_sub_value(),
atomic_toggle(), atomic_toggle_value()
Synopsis:

```c
#include <atomic.h>

void atomic_clr( volatile unsigned * loc,
                 unsigned bits );
```

Arguments:

- **loc**: A pointer to the value that you want to clear bits in.
- **bits**: The bits that you want to clear.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_clr()` function is a thread-safe way of doing an `(*loc) &= ~bits` operation.

The `atomic_*( )` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*( )` functions.

The `atomic_*( )` functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

Examples:

To safely clear the `0x10101010` bits in a flag:

```c
#include <atomic.h>
...
```
volatile unsigned flags;
...
atomic_clr( &flags, 0x10101010 );

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: Yes
- Signal handler: Yes
- Thread: Yes

**See also:**

atomic_add(), atomic_add_value(), atomic_set(), atomic_set_value(), atomic_sub(), atomic_sub_value(), atomic_toggle(), atomic_toggle_value()
atomic_clr_value()  
Safely clear a variable, returning the previous value

Synopsis:
```c
#include <atomic.h>

unsigned atomic_clr_value( volatile unsigned * loc,
        unsigned bits );
```

Arguments:
- `loc`  A pointer to the value that you want to clear bits in.
- `bits`  The bits that you want to clear.

Library:

```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_clr_value()` function is a thread-safe way of doing an `(*loc) &= ~bits` operation.

The `atomic_*( )` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*( )` functions.

The `atomic_*( )` functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

```
The atomic_clr_value() function may be slower than atomic_clr().
```
atomic_clr_value()

Returns:
The previous value of loc’s contents.

Examples:
To safely clear the 0x10101010 bits in a flag:

```
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...

previous = atomic_clr_value( &flags, 0x10101010 );
```

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
atomic_add(), atomic_add_value(), atomic_clr(), atomic_set(),
atomic_set_value(), atomic_sub(), atomic_sub_value(),
atomic_toggle(), atomic_toggle_value()
atomic_set()
Safely set bits in a variable

Synopsis:

```
#include <atomic.h>

void atomic_set( volatile unsigned * loc, 
                 unsigned bits );
```

Arguments:

- `loc` A pointer to the location whose bits you want to toggle.
- `bits` The bits that you want to set.

Library:

```
libc
```

Use the `-l` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_set()` function is a thread-safe way of doing an `(*loc) |= bits` operation.

The `atomic_*( )` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*( )` functions.

The `atomic_*( )` functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

Examples:

To safely set the 1 bit in a flag:

```
#include <atomic.h>
...
```
volatile unsigned flags;
...

atomic_set( &flags, 0x01 );

**Classification:**

QNX Neutrino

| Safety |  
|--------|--------|
| Cancellation point | No |
| Interrupt handler | Yes |
| Signal handler | Yes |
| Thread | Yes |

**See also:**

atomic_add(), atomic_add_value(), atomic_clr(), atomic_clr_value(), atomic_sub(), atomic_sub_value(), atomic_toggle(), atomic_toggle_value()
atomic_set_value()

Safely set bits in a variable, returning the previous value

Synopsis:

```c
#include <atomic.h>

unsigned atomic_set_value( volatile unsigned * loc,
                           unsigned bits );
```

Arguments:

- `loc` A pointer to the location whose bits you want to toggle.
- `bits` The bits that you want to set.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_set_value()` function is a thread-safe way of doing an `(*loc) |= bits` operation.

The `atomic_*()` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*()` functions.

The `atomic_*()` functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

The `atomic_set_value()` function may be slower than `atomic_set()`. 
atomic_set_value()

Returns:

The previous value of loc’s contents.

Examples:

To safely set the 1 bit in a flag:

```c
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...
previous = atomic_set_value( &flags, 0x01 );
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

atomic_add(), atomic_add_value(), atomic_clr(),
atomic_clr_value(), atomic_set(), atomic_sub(),
atomic_sub_value(), atomic_toggle(), atomic_toggle_value()
atomic_sub()

Safely subtract from a variable

Synopsis:

```c
#include <atomic.h>

void atomic_sub( volatile unsigned * loc,
                 unsigned decr);
```

Arguments:

`loc`  
A pointer to the value that you want to subtract from.

`decr`  
The number that you want to subtract.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `atomic_sub()` function is a thread-safe way of doing a `(*loc) -= decr` operation, even in a symmetric-multiprocessing system.

The `atomic_*()` functions are guaranteed to complete without being preempted by another thread.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*()` functions.

The `atomic_*()` functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

Examples:

Safely subtract 1 from a counter:

```c
#include <atomic.h>
...
```
atomic_sub()

```c
volatile unsigned count;
...
atomic_sub( &count, 1 );
```

**Classification:**
QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atomic_add(), atomic_add_value(), atomic_clr(), atomic_clr_value(), atomic_set(), atomic_set_value(), atomic_sub_value(), atomic_toggle(), atomic_toggle_value()`
atomic_sub_value()  
Safely subtract from a variable, returning the previous value

Synopsis:
#include <atomic.h>

unsigned atomic_sub_value( volatile unsigned * loc,  
                           unsigned decr );

Arguments:
loc   A pointer to the value that you want to subtract from.
decr  The number that you want to subtract.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The atomic_sub_value() function is a thread-safe way of doing a  
(*loc) -= decr operation, even in a symmetric-multiprocessing system.
The atomic_*() functions are guaranteed to complete without being preempted by another thread.
When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the atomic_*() functions.
The atomic_*() functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

---

The atomic_sub_value() function may be slower than atomic_sub().
atomic_sub_value()

Returns:
The previous value of loc’s contents.

Examples:
Safely subtract 1 from a counter:

```c
#include <atomic.h>
...
volatile unsigned count;
unsigned previous;
...
previous = atomic_sub_value( &count, 1 );
```

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
atomic_add(), atomic_add_value(), atomic_clr(), atomic_clr_value(), atomic_set(), atomic_set_value(), atomic_sub(), atomic_toggle(), atomic_toggle_value()
atomic_toggle()
Safely toggle a variable

Synopsis:
#include <atomic.h>

void atomic_toggle( volatile unsigned * loc, 
                   unsigned bits );

Arguments:
loc A pointer to the location whose bits you want to toggle.
bits The bits that you want to change.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The atomic_toggle() function is a thread-safe way of doing an
(loc) ^= bits operation.
The atomic_*( ) functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system.
When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the atomic_*( ) functions.
The atomic_*( ) functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

Examples:
To safely toggle the 0xdeadbeef bits in a flag:

#include <atomic.h>
...

September 10, 2007 QNX Neutrino Functions and Macros 229
volatile unsigned flags;
...

atomic_toggle( &flags, 0xdeadbeef );

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

atomic_add(), atomic_add_value(), atomic_clr(), atomic_clr_value(), atomic_set(), atomic_set_value(), atomic_sub(),
atomic_sub_value(), atomic_toggle_value()
**atomic toggle value()**

Safely toggle a variable, returning the previous value

**Synopsis:**

```c
#include <atomic.h>

unsigned atomic_toggle_value(
    volatile unsigned * loc,
    unsigned bits);
```

**Arguments:**

- `loc` A pointer to the location whose bits you want to toggle.
- `bits` The bits that you want to change.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `atomic_toggle_value()` function is a thread-safe way of doing an `(*loc) ^= bits` operation.

The `atomic_*()` functions are guaranteed to complete without being preempted by another thread, even in a symmetric-multiprocessing system.

When modifying a variable shared between a thread and an interrupt, you must either disable interrupts or use the `atomic_*()` functions.

The `atomic_*()` functions are also useful for modifying variables that are referenced by more than one thread (that aren’t necessarily in the same process) without having to use a mutex.

---

The `atomic_toggle_value()` function may be slower than `atomic_toggle()`. 

---
Returns:
The previous value of loc’s contents.

Examples:
To safely toggle the 0xdeadbeef bits in a flag:

```c
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...
previous = atomic_toggle_value( &flags, 0xdeadbeef );
```

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
atomic_add(), atomic_add_value(), atomic_clr(),
atomic_clr_value(), atomic_set(), atomic_set_value(), atomic_sub(),
atomic_sub_value(), atomic_toggle()
Synopsis:

```c
auxv_t * _auxv;
```

Description:

This global variable holds a pointer to a vector of auxiliary arguments to `main()`. For more information, see `<sys/auxv.h>`.

This variable isn’t defined in any header file. If you want to refer to it, you need to add your own `extern` statement.

Classification:

QNX Neutrino

See also:

`_argc, _argv, getopt(), main()`
The `basename()` function takes the pathname pointed to by `path` and returns a pointer to the final component of the pathname, deleting any trailing “/” characters.

The `basename()` function returns:

A pointer to the string “/”
  If the string consists entirely of the “/” character

A pointer to the string “.”
  If `path` is a NULL pointer, or points to an empty string

Together the `dirname()` and `basename()` functions yield a complete pathname. The expression `dirname(path)` obtains the pathname of the directory where `basename(path)` is found.
The `basename()` function might modify the string pointed to by `path`, and can return a pointer to static storage.

**Returns:**

A pointer to the final component of `path`.

**Examples:**

```c
#include <stdio.h>
#include <libgen.h>
#include <stdlib.h>

int main( int argc, char** argv )
{
    int x;
    for( x = 1; x < argc; x++ ) {
        printf( "%s\n", basename( argv[x] ) );
    }
    return EXIT_SUCCESS;
}
```

The table below shows the output of the program, given the input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/usr/lib</code></td>
<td><code>lib</code></td>
</tr>
<tr>
<td><code>/usr/</code></td>
<td><code>usr</code></td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>/</code></td>
</tr>
</tbody>
</table>

**Classification:**

POSIX 1003.1 XSI
basename() © 2007, QNX Software Systems GmbH & Co. KG.

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

dirname()
Synopsis:

```c
#include <strings.h>

int bcmp( const void *s1,
          const void *s2,
          size_t n );
```

Arguments:

- `s1`, `s2` The strings you want to compare.
- `n` The number of bytes to compare.

Library:

```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `bcmp()` function compares the byte string pointed to by `s1` to the string pointed to by `s2`. The number of bytes to compare is specified by `n`. NUL characters may be included in the comparison.

This function is similar to the ANSI `memcmp()` function, but tests only for equality. New code should use the ANSI function.

Returns:

- 0 The byte strings are identical.
- 1 The byte strings aren’t identical.
**Examples:**

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
    if( bcmp( "Hello there", "Hello world", 6 ) ) {
        printf( "Not equal\n" );
    } else {
        printf( "Equal\n" );
    }
    return EXIT_SUCCESS;
}
```

produces the output:

```
Equal
```

**Classification:**

POSIX 1003.1 XSI

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`bcopy()`, `bzero()`, `memcmp()`, `strcmp()`
Synopsis:

```c
#include <strings.h>

void bcopy( const void *src, 
    void *dst, 
    size_t n );
```

Arguments:

- `src` The string you want to copy.
- `dst` An existing array into which you want to copy the string.
- `n` The number of bytes to copy.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `bcopy()` function copies the byte string pointed to by `src` (including any NUL characters) into the array pointed to by `dst`. The number of bytes to copy is specified by `n`. Copying of overlapping objects is guaranteed to work properly.

This function is similar to the ANSI `memmove()` function, but the order of arguments is different. New code should use the ANSI function.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main( void )
{
```
auto char buffer[80];

bcopy( "Hello ", buffer, 6 );
bcopy( "world", &buffer[6], 6 );
printf( "%s\n", buffer );
return EXIT_SUCCESS;
}

produces the output:

Hello world

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

bcmp(), bzero(), memmove(), strcpy()
bind()
Bind a name to a socket

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

Arguments:

- **s**: The file descriptor to be bound.
- **name**: A pointer to the `sockaddr` structure that holds the address to be bound to the socket. The socket length and format depend upon its address family.
- **namelen**: The length of the `sockaddr` structure pointed to by `name`.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

When a socket is created with `socket()`, it exists in a namespace (address family) but has no name assigned to it. The `bind()` function assigns a name to that unnamed socket.

The `bind()` function assigns a local address. Use `connect()` to assign a remote address.

The rules used for binding names vary between communication domains.
Returns:

0     Success.
-1    An error occurred (errno is set).

Errors:

EACCES  The requested address is protected, and the current user
        has inadequate permission to access it.
EADDRINUSE
        The specified address is already in use.
EADDRNOTAVAIL
        The specified address isn’t available from the local
        machine.
EBADF   Invalid descriptor s.
EFAULT  The name parameter isn’t in a valid part of the user
        address space.
EINVAL  The socket is already bound to an address.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

ICMP, IP, TCP, and UDP protocols

connect(), getsockname(), listen(), socket()
bindresvport()  
Bind a socket to a privileged IP port

Synopsis:

```c
#include <sys/types.h>
#include <netinet/in.h>

int bindresvport(int sd,  
                 struct sockaddr_in * sin);
```

Arguments:

- `sd` The socket descriptor to bind to the port.
- `sin` A pointer to a `sockaddr_in` structure that specifies the privileged IP port.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `bindresvport()` function binds a socket descriptor to a privileged IP port (i.e. a port number in the range 0-1023).

- Only `root` can bind to a privileged port; this call fails for any other user.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).

Errors:

- `EACCES` You must be `root` to call `bindresvport()`.
- `EADDRINUSE` The specified address is already in use.
bindresvport()

EADDRNOTAVAIL
    The specified address isn’t available from the local machine.
EBADF    Invalid descriptor \(sd\).
EFAULT   The \(sin\) parameter isn’t a valid pointer to a \sockaddr_in\ structure.
EINVAL   The socket is already bound to a port.
EPFNOSUPPORT
    The protocol family isn’t supported.

Classification:
Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

\textit{connect()}, \textit{getsockname()}, \textit{listen()}, \textit{socket()}
brk()

Change the amount of space allocated for the calling process’s data segment

Synopsis:

```c
#include <unistd.h>

int brk( void* endds );
```

Arguments:

`endds` A pointer to the new end of the data segment.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `brk()` function is used to change dynamically the amount of space allocated for the calling process’s data segment (see the `exec*` functions).

The change is made by resetting the process’s break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process, its contents are undefined.

When a program begins execution using `execve()`, the break is set at the highest location defined by the program and data storage areas.

You can call `getrlimit()` to determine the maximum permissible size of the data segment; it isn’t possible to set the break beyond the `rlim_max` value returned from `getrlimit()`, i.e:

`end + rlim.rlim_max`
**Returns:**

0    Success.
-1   An error occurred (errno is set).

**Errors:**

ENOMEM  This could mean:
- The data segment size limit, as set by `setrlimit()`, would be exceeded.
- The maximum possible size of a data segment (compiled into the system) would be exceeded.
- Insufficient space exists in the swap area to support the expansion.
- Out of address space; the new break value would extend into an area of the address space defined by some previously established mapping (see `mmap()`).

EAGAIN  The total amount of system memory available for private pages is temporarily insufficient. This may occur even though the space requested was less than the maximum data segment size.

**Classification:**

Legacy Unix

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The behavior of `brk()` is unspecified if an application also uses any other memory functions (such as `malloc()`, `mmap()`, and `free()`). The `brk()` function has been used in specialized cases where no other memory allocation function provided the same capability. Use `mmap()` instead because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

The value of the argument to `brk()` is rounded up for alignment with eight-byte boundaries.

Setting the break may fail due to a temporary lack of swap space. It isn’t possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting `getrlimit()`.

See also:

`_btext`, `_edata`, `_end`, `_etext`, `exec()`, `execve()`, `execvp()`, `free()`, `getrlimit()`, `malloc()`, `mmap()`, `sbrk()`
**Synopsis:**

```c
#include <stdlib.h>

void *bsearch( const void *key,  
                const void *base,  
                size_t num,  
                size_t width,  
                int (*compar)( const void *pkey,  
                               const void *pbase) );
```

**Arguments:**

- **key**
  The object to search for.

- **base**
  A pointer to the first element in the array.

- **num**
  The number of elements in the array.

- **width**
  The size of an element, in bytes.

- **compare**
  A pointer to a user-supplied function that `lfind()` calls to compare an array element with the `key`.

  The arguments to the comparison function are:
  
  - **pkey** — the same pointer as `key`
  - **pbase** — a pointer to an element in the array.

  The comparison function must return an integer less than, equal to, or greater than zero if the `key` object is less than, equal to, or greater than the element in the array.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**bsearch()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The `bsearch()` function performs a binary search on the sorted array of `num` elements pointed to by `base`, for an item that matches the object pointed to by `key`.

**Returns:**

A pointer to a matching member of the array, or NULL if a matching object couldn’t be found.

If there are multiple values in the array that match the `key`, the return value could be any of these duplicate values.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static const char *keywords[] = {
    "auto",
    "break",
    "case",
    "char",
    /* ... */
    "while"
};

#define NUM_KW sizeof(keywords) / sizeof(char *)

int kw_compare( const void *p1, const void *p2 )
{
    const char *plc = (const char *) p1;
    const char **p2c = (const char **) p2;
    return( strcmp( plc, *p2c ) );
}

int keyword_lookup( const char *name )
{
    const char **key;

    key = (char const **) bsearch( name, keywords,
                                   NUM_KW, sizeof( char * ), kw_compare );
    if( key == NULL ) return( -1 );
```
bsearch()

    return key - keywords;
}

int main( void )
{
    printf( "%d\n", keyword_lookup( "case" ) );
    printf( "%d\n", keyword_lookup( "crigger" ) );
    printf( "%d\n", keyword_lookup( "auto" ) );

    return EXIT_SUCCESS;
}

This program produces the following output:

    2
   -1
    0

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

lfind(), lsearch(), qsort()
Synopsis:

N/A

Description:

This linker symbol defines the beginning of the text segment. This variable isn’t defined in any header file.

Classification:

QNX Neutrino

See also:

brk(), _edata, _end, _etext, sbrk()
Synopsis:

```c
#include <wchar.h>

wint_t btowc( int c );
```

Arguments:

- `c` The single-byte character that you want to convert.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `btowc()` function converts the given character (if it’s a valid one-byte character in the initial shift state) into a wide character. This function is the single-byte version of `mbtowc()`.

Returns:

The wide-character representation of the character, or WEOF if `c` has the value EOF or (unsigned char) `c` isn’t a valid one-byte character in the initial conversion state.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued…*
Safety

| Thread | Yes |

See also:

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```c
#include <strings.h>

void bzero( void *dst, 
   size_t n );
```

Arguments:

- `dst` An existing object that you want to fill with zeroes.
- `n` The number of bytes to fill.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `bzero()` function fills the first `n` bytes of the object pointed to by `dst` with zero (NUL) bytes.

This function is similar to the ANSI `memset()` function. New code should use the ANSI function.

Examples:

```c
#include <stdlib.h>
#include <string.h>

int main( void )
{
   char buffer[80];

   bzero( buffer, 80 );
   return EXIT_SUCCESS;
}
```
**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*bcmp(), bcopy(), memset(), strset()*
**Synopsis:**

```c
#include <math.h>

struct __cabsargs {
    double x; /* real part */
    double y; /* imaginary part */
};

double cabs( struct __cabsargs value );

struct __cabsfargs {
    float x; /* real part */
    float y; /* imaginary part */
};

float cabsf( struct __cabsfargs value );
```

**Arguments:**

- `value` The complex value that you want to get the absolute value of.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the absolute value of the complex number specified by `value`, using a calculation equivalent to:

```c
sqrt( ( value.x * value.x ) + ( value.y * value.y ) );
```

**Returns:**

The absolute value of `value`. 
Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

struct __cabsargs c = { -3.0, 4.0 };

int main( void )
{
    printf( "%f\n", cabs( c ) );
    return EXIT_SUCCESS;
}
```

produces the output:

5.000000

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`abs(), fabs(), labs()`
Synopsis:

```c
int cache_fini(struct cache_ctrl *cinfo);
```

Arguments:

- `cinfo` A pointer to the structure that was originally passed to `cache_init()`. See the `cache_init()` function.

Library:

- `libcach`e

Use the `-l cache` option to `qcc` to link against this library.

Description:

Call the `cache_fini()` function to free up any resources that were allocated by the `cache_init()` function. The `cinfo` is a pointer to the structure that was originally passed to `cache_init()`.

This function was added in the QNX Momentics 6.3.0 SP2.

Returns:

0

Classification:

QNX Neutrino

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes

continued...
**cache_fin()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`CACHE_FLUSH()`, `cache_init()`, `CACHE_INVAL()`
CACHE_FLUSH()
Flush cache lines associated with a data buffer

Synopsis:

```c
#include <sys/cache.h>

CACHE_FLUSH(struct cache_ctrl *cinfo,
    void *vaddr,
    uint64_t paddr,
    size_t len);
```

Arguments:

- `cinfo` A pointer to the structure that was initially passed to `cache_init()`.
- `vaddr` The virtual address of the buffer; this is a pointer to the data in the driver’s virtual address space.
- `paddr` The physical address of the buffer: this is typically in the same address space that the external device will use to reference the data. The physical address is obtained by calling `mem_offset64()`. Since this function is fairly costly, drivers typically allocate a pool of data buffers at initialization (e.g. by calling `mmap()` with the MAP_PHYS and MAP_ANON flags), and predetermine the physical addresses of the data.
- `len` The number of bytes in the buffer, for which cached data should be flushed to memory.

Library:

```
libcache
```

Use the `-l cache` option to `qcc` to link against this library.

Description:

This macro is used to flush any cache lines associated with a data buffer out to memory. This routine ensures that any modifications that have been made to the data by the CPU will be reflected by the contents of memory, and thus an external device reading the data.
won’t retrieve stale data. For more information about cache coherency, see cache_init().

Environment Variables

The following environment variables, if they exist, affect the behavior of this cache coherency function:

CACHE_NOP

Instructs the library that the CACHE_FLUSH() and CACHE_INVAL() macros should have no effect.

CACHE_MSYNC

Instructs the library that the CACHE_FLUSH() and CACHE_INVAL() macros should use the msync() C library call to perform cache synchronization.

This function was added in the QNX Momentics 6.3.0 SP2.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

CACHE_FLUSH() is implemented as a macro.
See also:

\textit{cache\_fini()}, \textit{cache\_init()}, \textit{CACHE\_INVAL()}
Register with the cache-coherency library

Synopsis:

```c
#include <sys/cache.h>

int cache_init(int flags,
               struct cache_ctrl *cinfo,
               const char *dllname);
```

Arguments:

- `flags` Zero or flags that control the behavior of the cache-coherency library. The only flag currently defined is:
  - `CACHE_INIT_FLAG_IGNORE_SCAN` — Specify that memory accesses to and from the device aren’t snooped, whether or not the data caches in the system are reported as snooping or not. This might be required for devices in the system that bypass the bus-snooping mechanism, or as a workaround for hardware bugs.

- `cinfo` A pointer to a structure that contains state information. The library uses this structure to store various information that uses when performing synchronization operations. The driver should allocate and initialize this structure. All of the members of the structure should be initialized with zeros, with the exception of the `fd` field. For more information regarding the members of this structure, see the description section.

- `dllname` The path of a DLL to load that provides cache-synchronization routines or NULL to use the default routines.

Library:

- `libcache`

Use the `-l cache` option to `qcc` to link against this library.
Description:

The `cache_init()` function initializes the cache coherency library (`libcache`). Your driver must call `cache_init()` before using the library.

Members of the structure

`cache_line_size`

When this function returns, this field will specify the size, in bytes, of a cache line’s worth of data. If the system implements a bus-snooping protocol, this field may contain zero.

`cache_flush_rate`

Provides a runtime indication to the driver, of the cost of flushing the cache:

- **CACHE_OP_RATE_SNOOP**
  
  Due to a bus-snooping mechanism, a cache flush operation has negligible cost.

- **CACHE_OP_RATE_INLINE**
  
  Cache flush operations are implemented with CPU-specific inline routines, and are inexpensive.

- **CACHE_OP_RATE_CALLOUT**
  
  Cache flush operations are implemented by calling an external function, which incurs a small CPU overhead.

- **CACHE_OP_RATE_MSYNC**
  
  Cache flush operations are implemented by calling the `msync()` function. Since this function is implemented with a system call, the operation will be very expensive. It is very unlikely that the library will end up calling `msync()`, but in the event that it does, the driver could potentially achieve better performance by mapping data buffers as noncacheable, so that it can avoid having to perform cache synchronization operations.
cache_init()

cache_invalidate_rate

 Provides a runtime indication to the driver of the cost of invalidating the cache. The defined values for this field are similar to those defined for the cache_flush_rate field.

fd  The driver should set this field to NOFD.

Other fields in the structure should not be referenced or modified.

Cache coherency

Device drivers for hardware that performs Direct Memory Accesses (DMA) use this cache coherency library. These devices are either bus-mastering devices that can directly read or modify memory, or devices that use a DMA controller to transfer data between the device and memory. The key factor is that memory may be accessed by an agent other than the CPU(s).

On some systems, cache coherency is performed by a bus-snooping protocol that is implemented in hardware. In such systems, the CPU(s) snoop all transactions on the memory bus and automatically keep their caches in sync with the memory.

For example, if an external agent modifies data at a given memory location, the CPU will observe the memory write cycle, and if it has a cached copy of the data at that memory location, it will invalidate the corresponding cache entry. The next time the CPU tries to examine the memory location, it will fetch the modified data from memory, instead of retrieving stale data from the cache.

Similarly, special action is taken if an external agent attempts to read a memory location, and a CPU has modified the memory location, but the modified copy of the data is in its cache, and hasn’t yet been written to memory. In this case, the read cycle is suspended while the CPU copies the updated version of the data out to memory. Once memory has been updated with the modified version, the read cycle can continue, and the external agent gets the updated copy of the data.

On other systems, where there is no such snooping protocol implemented in hardware, cache coherency between the CPU and
external devices must be maintained by driver software. These are typically single-CPU systems, since on SMP systems, bus-snooping is the usual mechanism of keeping the CPUs in sync with each other. To work on these systems, special action needs to be taken by driver software, to ensure data coherency between the CPU and external devices.

A driver ensures data coherency for systems that don’t have a bus-snooping protocol in different ways. The first one is the “big hammer” approach that simply disables caching for any memory location that can be accessed by both the CPU and the external device. This approach, however, has a severe performance penalty; for some network devices on certain systems, the throughput reduces to roughly half of the original value.

You can solve the above throughput problem by using cacheable data buffers, but perform synchronization operations on the data buffers at strategic points in the driver. For example, in the case of packet transmission for a network device, the driver must ensure that any data pertaining to the packet had been flushed out to memory, before allowing the DMA agent to begin copying the data. In the case of packet reception, the driver should invalidate any cached data pertaining to the packet buffer, before letting the DMA agent transfer data into the receive buffer. This eliminates the possibility that the CPU could write a cached portion of the data out to the memory buffer, after the network device had updated the buffer with new packet data.

The driver can perform cache flushing and invalidation operations in one of two ways. It can issue special CPU-dependent instructions that operate on the cache, or it can use the cache-coherency library (libcache). The latter approach is preferable, since it makes your driver portable. The library performs the correct thing based on the type of CPU it’s running on. For maximum portability, the library must be used whether the system has a bus-snooping protocol or not. If the system implements a bus-snooping protocol, the library determines this, and ensures that there are no unnecessary synchronization operations being performed.
This function was added in the QNX Momentics 6.3.0 SP2.

Returns:

0    Success.
-1   Failure and errno is set. If this function fails, it isn’t safe for devices to DMA to or from cacheable buffers. Additionally, calling other functions with the cache_ctrl structure that was provided will have unpredictable results.

Classification:

<table>
<thead>
<tr>
<th>QNX Neutrino</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

The cache-invalidation operation could have certain negative side effects, which the driver must take measures to avoid. If a data buffer partially shares a cache line with some other piece of data (including another data buffer), data corruption could occur. Since the invalidation is performed with cache-line granularity, invalidating data at the start or end of the buffer could potentially invalidate important data, such as program variables, which means that changes made to the data by the CPU could be inadvertently lost.

You can avoid this by padding the data buffers. The driver should pad the start of the buffer, so that it starts on the next cache line boundary. It should also pad the buffer out to the end of the last cache line of the
buffer. To do this, the driver can use the `cache_line_size` field of the `cache_ctrl` structure. Note that this value could be zero (e.g. if there is a cache-coherency protocol implemented in hardware), in which case the driver doesn’t need to do any padding.

See also:

`cache_fini()`, `CACHE_FLUSH()`, `CACHE_INVAL()`
Invalidate cache lines associated with a data buffer

Synopsis:

```c
#include <sys/cache.h>

CACHE_INVAL(struct cache_ctrl *cinfo,
            void *vaddr,
            uint64_t paddr,
            size_t len);
```

Arguments:

- `cinfo`: A pointer to the structure that was initially passed to `cache_init()`.
- `vaddr`: The virtual address of the buffer; this is a pointer to the data in the driver’s virtual address space.
- `paddr`: The physical address of the buffer: this is typically in the same address space that the external device will use to reference the data. The physical address is obtained by calling `mem_offset64()`.
- `len`: The number of bytes in the buffer, for which the associated cache lines should be invalidated.

Library:

- `libcachef`

Use the `-l cache` option to `qcc` to link against this library.

Description:

This macro is used to invalidate any cache lines associated with a data buffer. This routine ensures that any subsequent modifications that are made to the data by an external device will not be corrupted by the CPU writing back cached data to the memory, and ensures that once
the data has been modified, the CPU will fetch the updated data from memory, instead of retrieving stale data from the cache.

Environment Variables

The following environment variables, if they exist, affect the behavior of this cache coherency function:

CACHE_NOP

Instructs the library that the CACHE_FLUSH() and CACHE_INVALID() macros should have no effect.

CACHE_MSYNC

Instructs the library that the CACHE_FLUSH() and CACHE_INVALID() macros should use the msync() C library call to perform cache synchronization.

For more information about cache coherency, see cache_init().

This function was added in the QNX Momentics 6.3.0 SP2.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

\textit{CACHE\_INVAL()} is implemented as a macro.

See also:

\textit{cache\_fini()}, \textit{CACHE\_FLUSH()}, \textit{cache\_init()}
calloc()
Allocate space for an array

Synopsis:
```c
#include <stdlib.h>

void* calloc ( size_t n,
               size_t size );
```

Arguments:

- `n` The number of array elements to allocate.
- `size` The size, in bytes, of one array element.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `calloc()` function allocates space from the heap for an array of `n` objects, each of `size` bytes, and initializes them to 0.

A block of memory allocated with the `calloc()` function should be freed using the `free()` function.

Returns:

A pointer to the start of the allocated memory, or NULL if an error occurred (`errno` is set).

Errors:

- `ENOMEM` Not enough memory.
- `EOK` No error.
**cdecl()**

© 2007, QNX Software Systems GmbH & Co. KG.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>

int main( void ) {
    char* buffer;

    buffer = (char* )calloc( 80, sizeof(char) );
    if( buffer == NULL ) {
        printf( "Can’t allocate memory for buffer!\n" );
        return EXIT_FAILURE;
    }

    free( buffer );

    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`free()`, `malloc()`, `realloc()`, `sbrk()`
cbrt(), cbrtf()
Compute the cube root of a number

Synopsis:

```c
#include <math.h>

double cb rt ( double x );

float cb rtf ( float x );
```

Arguments:

- `x`  
The number whose cube root you want to calculate.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `cb rt()` and `cb rtf()` functions compute the cube root of `x`.

Returns:

The cube root of `x`. If `x` is NAN, `cb rt()` returns NAN.

Examples:

```c
#include <stdio.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv) {
  double a, b;
  
a = 27.0;
  b = cb rt(a);
  printf("The cube root of %f is %f \n", a, b);
  
  return(0);
}
```

produces the output:

```
The cube root of 27.000000 is 3.000000
```
cbrt(), cbrtf()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sqrt()
ceil(), ceilf()
Round up a value to the next integer

Synopsis:

```c
#include <math.h>

double ceil( double x );

float ceilf( float x );
```

Arguments:

- `x` The value you want to round.

Library:

- `libm`
  Use the `-l m` option to `qcc` to link against this library.

Description:

The `ceil()` and `ceilf()` functions round the value of `x` up to the next integer (rounding towards the “ceiling”).

Returns:

The smallest integer \( \geq x \).

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f %f %f %f\n", ceil( -2.1 ),
            ceil( -2. ), ceil( 0.0 ), ceil( 2. ),
            ceil( 2.1 ) );
}
```
return EXIT_SUCCESS;
}

produces the output:

-2.000000 -2.000000 0.000000 2.000000 3.000000

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`floor()`
**Synopsis:**

```c
#include <termios.h>

speed_t cfgetispeed(
    const struct termios* termios_p);
```

**Arguments:**

- `termios_p` : A pointer to a `termios` structure that describes the terminal’s control attributes.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `cfgetispeed()` function returns the input baud rate that’s stored in the `termios` structure pointed to by `termios_p`.

You can get a valid `termios` control structure for an opened device by calling `tcgetattr()`.

**Returns:**

The input baud rate stored in `*termios_p`, or `-1` if an error occurs (`errno` is set).

**Errors:**

- `EINVAL` : One of the arguments is invalid.
- `ENOTTY` : This function isn’t supported by the system.
Examples:

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "/dev/ser1", O_RDWR );
tcgetattr( fd, &termios_p);

    /*
     * Get input baud rate
     */
    speed = cfgetispeed( &termios_p);
    printf( "Input baud: %ld\n", speed );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

`errno`, `cfgetospeed()`, `cfgetispeed()`, `cfsetospeed()`, `tcgetattr()`, `tcsetattr()`, `termios`
$\text{cfgetospeed()}$  

© 2007, QNX Software Systems GmbH & Co. KG.

Return the output baud rate that’s stored in a $\text{termios}$ structure

**Synopsis:**

```c
#include <termios.h>

speed_t cfgetospeed(
    const struct termios* termios_p);
```

**Arguments:**

- $\text{termios}_p$  
  A pointer to a $\text{termios}$ structure that describes the terminal’s control attributes.

**Library:**

- libc

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The $\text{cfgetospeed()}$ function returns the output baud rate that’s stored in the $\text{termios}$ structure pointed to by $\text{termios}_p$.

You can get a valid $\text{termios}$ control structure for an opened device by calling $\text{tcgetattr()}$.

**Returns:**

The output baud rate stored in $\ast termios_p$, or -1 if an error occurs ($\text{errno}$ is set).

**Errors:**

- $\text{EINVAL}$  
  One of the arguments is invalid.

- $\text{ENOTTY}$  
  This function isn’t supported by the system.
Examples:

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "*/dev/ser1", O_RDWR );
    tcgetattr( fd , &termios_p);

    /*
     * Get output baud rate
     */
    speed = cfgetospeed( &termios_p);
    printf( "Output baud: %ld\n", speed );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

errno, cfgetispeed(), cfsetispeed(), cfsetospeed(), tcgetattr(),
tcsetattr(), termios
Synopsis:

```
#include <cfgopen.h>

int cfgopen( const char * path,
             unsigned flags,
             const char * historical,
             char * namebuf,
             int nblen );
```

Arguments:

- **path**: The name of the configuration file that you want to open.
- **flags**: Flags that control the opening; see below.
- **historical**: A optional file to open as a last resort if none of the criteria for finding the path is met. This string works like a path search order, and lets you search more than one location. You can also specify %H to substitute the hostname value into the string. Specify NULL to ignore this option.
- **namebuf**: A buffer to save the pathname in. Specify NULL to ignore this option.
- **nblen**: The length of the buffer pointed to by namebuf. Specify 0 to ignore this option.

Library:

- **libc**
  
  Use the -l c option to qcc to link against this library. This library is usually included automatically.
This function is in \texttt{libc.a}, but not in \texttt{libc.so} (in order to save space).

\section*{Description:}

The \texttt{cfgopen()} function opens the configuration file named by \textit{path}. This function is a cover function for \texttt{open()} that searches several default system locations for your files, based on specified characteristics.

The value of \texttt{flags} correspond to, and have similar limitations of, the standard \texttt{open()} flags. The \texttt{flags} value is constructed by the bitwise ORing of values from the following list, defined in the \texttt{<cfgopen.h>} header file. Applications must specify exactly one of these file-access modes in the value of \texttt{flag}:

\begin{itemize}
\item \texttt{CFGFILE\_RDONLY}
  \begin{itemize}
  \item Open for reading only.
  \end{itemize}
\item \texttt{CFGFILE\_RDWR}
  \begin{itemize}
  \item Open for reading and writing.
  \end{itemize}
\item \texttt{CFGFILE\_WRONLY}
  \begin{itemize}
  \item Open for writing only.
  \end{itemize}
\end{itemize}

You can also include any combination of these bits in the value of \texttt{flag}:

\begin{itemize}
\item \texttt{CFGFILE\_APPEND}
  \begin{itemize}
  \item If set, the file offset is set to the end of the file prior to each write.
  \end{itemize}
\item \texttt{CFGFILE\_CREAT}
  \begin{itemize}
  \item If the file doesn’t exist, it’s created with mode 0644, the file’s user ID is set to the effective user ID of the process, and the group ID is set to the effective group ID of the process or the group ID of the file’s parent directory (see \texttt{chmod()}).
  \end{itemize}
\end{itemize}
**cfgopen()**

**CFGFILE_EXCL**

If CFGFILE_EXCL and CFGFILE_CREAT are set, and the file exists, cfgopen() fails. The check for the existence of the file and the creation of the file if it doesn’t exist is atomic with respect to other processes attempting the same operation with the same filename. Specifying CFGFILE_EXCL without CFGFILE_CREAT has no effect.

**CFGFILE_TRUNC**

If the file exists and is a regular file, and the file is successfully opened CFGFILE_WRONLY or CFGFILE_RDWR, the file length is truncated to zero and the mode and owner are left unchanged. CFGFILE_TRUNC has no effect on FIFO or block or character special files or directories. Using CFGFILE_TRUNC with CFGFILE_RDONLY has no effect.

**Search condition flags**

In order to hint to the function where it should access or construct (in the case of CFGFILE_CREAT) path, there are several bits that you can specify and OR into flags. When specified, the bits are accessed using the following search order:

1. CFGFILE_USER_NODE

   

   | $HOME/ .cfg/node_name/path |

2. CFGFILE_USER

   

   | $HOME/ .cfg/path |

3. CFGFILE_NODE

   

   | /etc/host_cfg/node_name/path |

4. CFGFILE_GLOBAL

   

   | path |

where node_name is the value you get by calling confstr() for CS_HOSTNAME.
If the directory /etc/host_cfg doesn’t exist on the system, the following flags are transformed automatically:

- CFGFILE_USER_NODE becomes CFGFILE_USER
- CFGFILE_NODE becomes CFGFILE_GLOBAL

When creating a file or opening a file for writing, you can specify only one of the above location flags. Set CFGFILE_NOFD when you need only the pathname, not the file descriptor. If a directory path doesn’t exist when a file is opened for creation, cfgopen() attempts to create the path.

Returns:

A valid file descriptor if CFGFILE_NOFD isn’t specified, a nonnegative value if CFGFILE_NOFD is specified, or -1 if an error occurs.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
cfgopen()
Set terminal attributes

Synopsis:
```
#include <termios.h>

int cfmakeraw( struct termios * termios_p );
```

Arguments:
`termios_p` A pointer to a `termios` structure that describes the terminal’s control attributes.

Library:
`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `cfmakeraw()` function sets the terminal attributes as follows:

```
termios_p->c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL|IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHONL|ICANON|ISIG|IXTREM);
termios_p->c_cflag &= ~(CSIZE|PARENB);
termios_p->c_cflag |= CS8;
```

You can get a valid `termios` control structure for an opened device by calling `tcgetattr()`.

Returns:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (<code>errno</code> indicates the reason)</td>
</tr>
</tbody>
</table>

Classification:
Unix
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno`, `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `tcsetattr()`, `termios`
Synopsis:

```c
#include <malloc.h>

int cfree( void *ptr );
```

Arguments:

- `ptr` A pointer to the block of memory that you want to free. It’s safe to call `cfree()` with a NULL pointer.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `cfree()` function deallocates the memory block specified by `ptr`, which was previously returned by a call to `calloc()`, `malloc()` or `realloc()`.

Returns:

- 1

Classification:

- Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

Calling `cfree()` on a pointer already deallocated by a call to `cfree()`, `free()`, or `realloc()` could corrupt the memory allocator’s data structures.

See also:

`alloca()`, `calloc()`, `free()`, `malloc()`, `realloc()`, `sbrk()`
Set the input baud rate in a termios structure

Synopsis:
```
#include <termios.h>

int cfsetispeed( struct termios* termios_p, speed_t speed );
```

Arguments:
- `termios_p` A pointer to a termios structure that describes the terminal’s control attributes.
- `speed` The new speed. Valid values for `speed` are defined in `<termios.h>`.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `cfsetispeed()` function sets the input baud rate within the termios structure pointed to by `termios_p` to be `speed`.

You can get a valid termios control structure for an opened device by calling `tcgetattr()`.
The new baud rate isn’t effective until you call `tcsetattr()` with this modified `termios` structure.

- Attempts to set baud rates to values that aren’t supported by the hardware are ignored, and cause `tcsetattr()` to return an error, but `cfsetispeed()` doesn’t indicate an error.
- Attempts to set input baud rates to a value that’s different from the output baud rate, when the hardware doesn’t support split baud rates, cause the input baud rate to be ignored, but no error is generated.

**Returns:**

- 0 Success.
- -1 An error occurred (`errno` is set).

**Errors:**

- `EINVAL` One of the arguments is invalid.
- `ENOTTY` This function isn’t supported by the system.

**Examples:**

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "*/dev/ser1", O_RDWR );
    tcgetattr( fd, &termios_p );

    /*
     * Set input baud rate
     */
```
cfsetispeed()

        */
        speed = 9600;
        cfsetispeed( &termios_p, speed );
        tcsetattr( fd, TCSADRAIN, &termios_p);

        close( fd);
        return EXIT_SUCCESS;
    }

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

errno, cfgetispeed(), cfgetospeed(), cfsetospeed(), tcgetattr(),
tcsetattr(), termios
Synopsis:

```c
#include <termios.h>

int cfsetospeed( struct termios *termios_p, speed_t speed );
```

Arguments:

- `termios_p` A pointer to a `termios` structure that describes the terminal’s control attributes.
- `speed` The new speed. Valid values for `speed` are defined in `<termios.h>`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `cfsetospeed()` function sets the output baud rate within the `termios` structure pointed to by `termios_p` to be `speed`.

You can get a valid `termios` control structure for an opened device by calling `tcgetattr()`.

- The new baud rate isn’t effective until you call `tcsetattr()`, with this modified `termios` structure.
- Attempts to set baud rates to values that aren’t supported by the hardware are ignored, and cause `tcsetattr()` to return an error, but `cfsetospeed()` doesn’t indicate an error.

Setting the output baud rate to B0 causes the connection to be dropped. If `termios_p` represents a modem, the modem control lines will be turned off.
**Returns:**

0    Success.

-1    An error occurred (errno indicates the reason).

**Errors:**

EINV AL One of the arguments is invalid.

ENOTTY This function isn’t supported by the system.

**Examples:**

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;
    struct termios termios_p;
    speed_t speed;

    fd = open( "/dev/ser1", O_RDWR );
    tcgetattr( fd, &termios_p);

    /*
    *   Set output baud rate
    */
    speed = B9600;
    cfsetospeed( &termios_p, speed );
    tcsetattr( fd, TCSADRAIN, &termios_p);

    close( fd );
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, cfgetispeed(), cfgetospeed(), cfsetispeed(), tcgetattr(), tcsetattr(), termios
Create a communications channel

Synopsis:

```c
#include <sys/neutrino.h>

int ChannelCreate( unsigned flags );

int ChannelCreate_r( unsigned flags );
```

Arguments:

- `flags` Flags that can be used to request notification pulses from the kernel or request other changes in behavior; a combination of the following:
  - `_NTO_CHF_COID_DISCONNECT`
  - `_NTO_CHF_DISCONNECT`
  - `_NTO_CHF_FIXED_PRIORITY`
  - `_NTO_CHF_NET_MSG`
  - `_NTO_CHF_REPLY_LEN`
  - `_NTO_CHF_SENDER_LEN`
  - `_NTO_CHF_THREAD_DEATH`
  - `_NTO_CHF_UNBLOCK`
  
  For more information, see below.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ChannelCreate()` and `ChannelCreate_r()` kernel calls create a channel that can be used to receive messages and pulses. Once created, the channel is owned by the process and isn’t bound to the creating thread.
These functions are identical, except in the way they indicate errors. See the Returns section for details.

Threads wishing to communicate with the channel attach to it by calling `ConnectAttach()`. The threads may be in the same process, or in another process on the same node (or a remote node if the network manager is running). Once attached, these threads use `MsgSendv()` or `MsgSendPulse()` to enqueue messages and pulses on the channel. Messages and pulses are enqueued in priority order.

To dequeue and read messages and pulses from a channel, use `MsgReceivev()`. Any number of threads may call `MsgReceivev()` at the same time, in which case they block and queue (if no messages or pulses are waiting) for a message or pulse to arrive. A multi-threaded I/O manager typically creates multiple threads and has them all RECEIVE-blocked on the channel.

The return value of `ChannelCreate()` is a channel ID, an `int` taken from a channel vector on the process. Most managers use a single channel for most, if not all, their communications with clients. Additional channels can be used as special channels for information.

By default, when a message is received from a channel, the thread priority of the receiver is set to match that of the thread that sent the message. This basic priority inheritance prevents priority inversion. If a message arrives at a channel and there’s no thread waiting to receive it, the system boosts (if necessary) all threads in the process that have received a message from the channel in the past. This boost prevents a priority inversion of the client in the case where all threads are currently working on behalf of other clients, perhaps at a lower priority. When a thread is first created, it isn’t associated with a channel until it does a `MsgReceivev()` on it. In the case of multiple channels, a thread is associated with the last channel it received from.

After receiving a message, a thread can dissociate itself from the channel by calling `MsgReceivev()` with a -1 for the channel ID. Priority inheritance can be disabled by setting `_NTO_CHF_FIXED_PRIORITY` in the `flags` argument. In this case a thread’s priority isn’t affected by messages it receives on a channel.
A manager typically involves the following loop. There may be one or more threads in the loop at a time. Note that your program (not each thread) should call `ChannelCreate()` only once.

```c
iov_t iov;
...
SETIOV( &iov, &msg, sizeof( msg ) );
...
chid = ChannelCreate(flags);
...
for(;;) {
    /*
       Here's a one-part message; you could just as easily receive a 20-part message by filling in the iov appropriately.
    */
    rcvid = MsgReceivev(chid, &iov, 1, &info);
    /* msg is filled in by MsgReceivev() */
    switch(msg.type) {
        ...
    }
    /* iov could be filled in again to point to a new message */
    MsgReplyv(rcvid, iov, 1);
}
```

Some of the channel flags in the `flags` argument request changes from the default behavior; others request notification pulses from the kernel. The pulses are received by `MsgReceivev()` on the channel and are described by a `_pulse` structure.

The channel flags and (where appropriate) associated values for the pulse's `code` and `value` are described below.

**_NTO_CHF_COID_DISCONNECT**

Pulse code:       _PULSE_CODE_COIDDEATH

Pulse value:      Connection ID (coid) of a connection that was attached to a destroyed channel.

Deliver a pulse to this channel for each connection that belongs to the calling process when the channel that the connection is attached to is destroyed. Only one channel per process can have this flag set.
If a channel has one or both of \_NTO\_CHF\_COID\_DISCONNECT or \_NTO\_CHF\_THREAD\_DEATH set, neither flag may be set for any other channel in the process.

\_NTO\_CHF\_DISCONNECT

Pulse code: \_PULSE\_CODE\_DISCONNECT
Pulse value: None

Deliver a pulse when all connections from a process are detached (e.g. close(), ConnectDetach(), name_close()). If a process dies without detaching all its connections, the kernel detaches them from it. When this flag is set, the server must call ConnectDetach( scoid ) where scoid is the server connection ID in the pulse message. Failure to do so leaves an invalid server connection ID that can’t be reused. Over time, the server may run out of available IDs. If this flag isn’t set, the kernel removes the server connection ID automatically, making it available for reuse.

\_NTO\_CHF\_FIXED\_PRIORITY

Suppress priority inheritance when receiving messages. Receiving threads won’t change their priorities to those of the sending threads. If you’re using adaptive partitioning, the receiving threads won’t run in the sending threads’ partitions.

\_NTO\_CHF\_NET\_MSG

Reserved for the io_net resource manager.

\_NTO\_CHF\_REPLY\_LEN

Request that the length of the reply be included in the dstmsglen member of the \_msg\_info structure that MsgReceivev() fills in. The dstmsglen member is valid only if you set this channel flag when you create the channel.
ChannelCreate(), ChannelCreate_r() © 2007, QNX Software Systems GmbH & Co. KG.

_NTO_CHF_SENDER_LEN

Request that the length of the source message be included in the srcmsglen member of the _msg_info structure that MsgReceivev() fills in. The srcmsglen member is valid only if you set this channel flag when you create the channel.

_NTO_CHF_THREAD_DEATH

Pulse code: _PULSE_CODE_THREADDEATH
Pulse value: Thread ID (tid)

Deliver a pulse on the death of any thread in the process that owns the channel. Only one channel per process can have this flag set.

If a channel has one or both of _NTO_CHF_COID_DISCONNECT or _NTO_CHF_THREAD_DEATH set, neither flag may be set for any other channel in the process.

_NTO_CHF_UNBLOCK

Pulse code: _PULSE_CODE_UNBLOCK
Pulse value: Receive ID (rcvid)

In most cases, you’ll set the _NTO_CHF_UNBLOCK flag.

Deliver a pulse when a thread that’s REPLY-blocked on a channel attempts to unblock before its message is replied to. This occurs between the time of a MsgReceivev() and a MsgReplyv() by the server. The sending thread may be unblocked because of a signal or a kernel timeout.

If the sending thread unblocks, MsgReplyv() fails. The manager may not be in a position to handle this failure. It’s also possible that the client will die because of the signal and never send another message. If the manager is holding onto resources for the client (such as an
open file), it may want to receive notification that the client wants to break out of its `MsgSendv()`.

Setting the `_NTO_CHF_UNBLOCK` bit in `flags` prevents a thread that’s in the REPLY-blocked state from unblocking. Instead, a pulse is sent to the channel, informing the manager that the client wishes to unblock. In the case of a signal, the signal will be pending on the client thread. When the manager replies, the client is unblocked and at that point, any pending signals are acted upon. From the client’s point of view, its `MsgSendv()` will have completed normally and any signal will have arrived on the opcode following the successful kernel call.

When the manager receives the pulse, it can do one of these things:

- If it believes that it will be replying shortly, it can discard the pulse, resulting in a small latency in the unblocking, or it can signal the client. A short blocking request to a filesystem often takes this approach.

- If the reply is going to take some time or an unknown amount of time, the manager should cancel the current operation and reply back with an error or whatever data is available at this time in the reply message to the client thread. A request to a device manager waiting for input would take this approach.

**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

```
ChannelCreate()
```

The channel ID of the newly created channel. If an error occurs, the function returns -1 and sets `errno`.
ChannelCreate(), ChannelCreate_r() © 2007, QNX Software Systems GmbH & Co. KG.

ChannelCreate_r()

The channel ID of the newly created channel. This function does NOT set errno. If an error occurs, the function returns the negative of a value from the Errors section.

Errors:

EAGAIN All kernel channel objects are in use.
EBUSY The _NTO_CHF_COID_DISCONNECT or the _NTO_CHF_THREAD_DEATH flag was given and another channel belonging to this process already has the same flag set.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ChannelDestroy(), close(), ConnectAttach(), ConnectDetach(), _msg_info, MsgReceivev(), MsgReplyv(), MsgSendv(), MsgSendPulse(), name_close(), _pulse
ChannelDestroy(),
ChannelDestroy_r()
Destroy a communications channel

Synopsis:

```c
#include <sys/neutrino.h>

int ChannelDestroy( int chid );
int ChannelDestroy_r( int chid );
```

Arguments:

- **chid**: The channel ID, returned by ChannelCreate(), of the channel that you want to destroy.

Library:

**libc**

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

These kernel calls remove a channel specified by the channel ID `chid` argument. Once destroyed, any attempt to receive messages or pulses on the channel will fail. Any threads that are blocked on the channel by calling `MsgReceivev()` or `MsgSendv()` will be unblocked and return with an error.

The `ChannelDestroy()` and `ChannelDestroy_r()` functions are identical except in the way they indicate errors. See the Returns section for details.

When the channel is destroyed, all server connection IDs become invalid. The client connections are also marked invalid but remain in existence until the client removes them by calling `ConnectDetach()`.

An attempt by the client to use one of these invalid connections using `MsgSendv()` or `MsgSendPulse()` will return with an error.

A server typically destroys its channels prior to its termination. If it fails to do so, the kernel destroys them automatically when the process dies.
Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

`ChannelDestroy()`

If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

`ChannelDestroy_r()`

EOK is returned on success. This function does NOT set `errno`. If an error occurs, the function may return any value in the Errors section.

Errors:

EINVVAL The channel specified by `chid` doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`ChannelCreate()`, `MsgReceivev()`
**Synopsis:**

```c
#include <unistd.h>

int chdir( const char* path );
```

**Arguments:**

`path` The new current working directory.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `chdir()` function changes the current working directory to `path`, which can be relative to the current working directory or an absolute path name.

**Returns:**

0 Success.

-1 An error occurred (`errno` is set).

**Errors:**

- `EACCES` Search permission is denied for a component of `path`.
- `ELOOP` Too many levels of symbolic links or prefixes.
- `ENAMETOOLONG` The `path` argument is longer than `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.
- `ENOENT` The specified `path` doesn’t exist, or `path` is an empty string.
chdir()

ENOMEM There wasn’t enough memory to allocate a control structure.

ENOSYS The chdir() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of path is not a directory.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char* argv[] )
{
    if( argc != 2 ) {
        fprintf( stderr, "Use: cd <directory>\n" );
        return EXIT_FAILURE;
    }

    if( chdir( argv[1] ) == 0 ) {
        printf( "Directory changed to %s\n", argv[1] );
        return EXIT_SUCCESS;
    } else {
        perror( argv[1] );
        return EXIT_FAILURE;
    }
}
```

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

There’s only one current working directory per process. In a multithreaded application, any thread calling `chdir()` will change the current working directory for all threads in that process.

See also:

`errno`, `getcwd()`, `mkdir()`, `rmdir()`
**Synopsis:**

```c
#include <sys/types.h>
#include <sys/stat.h>

int chmod( const char * path, 
            mode_t mode );
```

**Arguments:**

- `path`: The name of the file whose permissions you want to change.
- `mode`: The new permissions for the file. For more information, see “Access permissions” in the documentation for `stat()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `chmod()` function changes S_ISUID, S_ISGID, S_ISVTX and the file permission bits of the file specified by the pathname pointed to by `path` to the corresponding bits in the `mode` argument. The application must ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges to do this.

If a directory is writable and the sticky bit (S_ISVTX) is set on the directory, a process can remove or rename a file within that directory only if one or more of the following is also true:

- The effective user ID of the process matches the file’s owner ID.
- The effective user ID of the process matches the directory’s owner ID.
- The file is writable by the effective user ID of the process.
- The user is a superuser (effective user ID of 0).
If a directory has the set-group ID bit set, a file created in that directory will have the same group ID as that directory. Otherwise, the newly created file’s group ID is set to the effective group ID of the creating process.

If the calling process doesn’t have appropriate privileges, and if the group ID of the file doesn’t match the effective group ID, and the file is a regular file, bit S_ISGID (set-group-ID on execution) in the file’s mode is cleared on a successful return from `chmod()`.

If the effective user ID of the calling process is equal to the file owner, or the calling process has appropriate privileges (for example, it belongs to the superuser), `chmod()` sets S_ISUID, S_ISGID and the file permission bits, defined in the `<sys/stat.h>` header file, from the corresponding bits in the `mode` argument. These bits define access permissions for the user associated with the file, the group associated with the file and all others.

This call has no effect on file descriptors for files that are already open.

If `chmod()` succeeds, the `st_ctime` field of the file is marked for update.

**Returns:**

0 Success.

-1 An error occurred (`errno` is set).

**Errors:**

- **EACCES** Search permission is denied on a component of the path prefix.

- **ELOOP** Too many levels of symbolic links or prefixes.

- **ENAMETOOLONG** The length of the path string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

- **ENOTDIR** A component of the path prefix isn’t a directory.
ENOENT  The file doesn’t exist, or the path arguments points to
  an empty string.

ENOSYS  The chmod() function isn’t implemented for the
  filesystem specified in path.

EPERM   The effective user ID doesn’t match the owner of the
  file, and the calling process doesn’t have appropriate
  privileges.

EROFS   The file resides on a read-only filesystem.

Examples:

/*
 * Change the permissions of a list of files
 * to by read/write by the owner only
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv )
{
  int i,
  int ecode = 0;

  for( i = 1; i < argc; i++ ) {
    if( chmod( argv[i], S_IRUSR | S_IWUSR ) == -1 ) {
      perror( argv[i] );
      ecode++;
    }
  }

  return ecode;
}

Classification:

  POSIX 1003.1
### chmod()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`chown()`, `errno`, `fchmod()`, `fchown()`, `fstat()`, `open()`, `stat()`
**Synopsis:**

```c
#include <sys/types.h>
#include <unistd.h>

int chown( const char * path, 
           uid_t owner, 
           gid_t group );
```

**Arguments:**

- `path` The name of the file whose ownership you want to change.
- `owner` The user ID of the new owner.
- `group` The group ID of the new owner.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `chown()` function changes the user ID and group ID of the file specified by `path` to be the numeric values contained in `owner` and `group`, respectively.

If the named file is a symbolic link, `chown()` changes the ownership of the file or directory to which the symbolic link refers; `lchown()` changes the ownership of the symbolic link file itself.

Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges (for example, the superuser) may change the ownership of a file.

In QNX Neutrino, the `_POSIX_CHOWN_RESTRICTED` flag (tested via the `_PC_CHOWN_RESTRICTED` flag in `pathconf()`), is enforced for `path`. This means that only the superuser may change the ownership or the group of a file to anyone. Normal users can’t give a
chown()

file away to another user by changing the file ownership, nor to another group by changing the group ownership.

If the path argument refers to a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode are cleared, if the function is successful.

If chown() succeeds, the st_ctime field of the file is marked for update.

Returns:

0 Success.

-1 (no changes were made in the user ID and group ID of the file).
   An error occurred (errno is set).

Errors:

EACCES Search permission is denied on a component of the path prefix.

ELOOP Too many levels of symbolic links or prefixes.

ENAMETOOLONG
   The length of the path string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT A component of the path prefix doesn’t exist, or the path arguments points to an empty string.

ENOSYS The chown() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of the path prefix isn’t a directory.

EPERM The effective user ID doesn’t match the owner of the file, or the calling process doesn’t have appropriate privileges.

EROF S The named file resides on a read-only filesystem.
**Examples:**

```c
/*
 * Change the ownership of a list of files
 * to the current user/group
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( int argc, char** argv )
{
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( chown( argv[i], getuid(), getgid() ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }
    exit( ecode );
}
```

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`chmod()`, `errno`, `fchown()`, `fstat()`, `lchown()`, `open()`, `stat()`
chroot()

Change the root directory

Synopsis:

```c
#include <unistd.h>

int chroot( const char *path );
```

Arguments:

`path` The name of the new root directory.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `chroot()` function causes the `path` directory to become the root directory, the starting point for path searches for path names beginning with `/`. The user’s working directory is unaffected.

The `..` entry in the root directory is interpreted to mean the root directory itself. Thus, you can’t use `..` to access files outside the subtree rooted at the root directory.

Returns:

0 Success.

-1 An error occurred; `errno` is set.

Errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Search permission is denied for a component of <code>path</code>.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The descriptor isn’t valid.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>path</code> argument points to an illegal address.</td>
</tr>
</tbody>
</table>
chroot()

EINTR  A signal was caught during the chroot() function.

EIO    An I/O error occurred while reading from or writing to the filesystem.

ELOOP  Too many symbolic links were encountered in translating path.

EMULTIHOP Components of path require hopping to multiple remote machines, and the filesystem type doesn’t allow it.

ENAMETOOLONG
The length of the path argument exceeds \{PATH_MAX\}, or the length of a path component exceeds \{NAME_MAX\} while \{_POSIX_NO_TRUNC\} is in effect.

ENOENT The named directory doesn’t exist or is a null pathname.

ENOLINK The path points to a remote machine and the link to that machine is no longer active.

ENOTDIR Any component of the path name isn’t a directory.

EPERM  The effective user of the calling process isn’t the superuser.

Classification:
 Legacy Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
chroot()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

chdir()
chsize()

Synopsis:

```c
#include <unistd.h>

int chsize( int filedes,
            long size );
```

Arguments:

- `filedes` A file descriptor for the file whose size you want to change.
- `size` The new size of the file, in bytes.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `chsize()` function extends or truncates the file specified by `filedes` to `size` bytes. The file is padded with NUL (‘\0’) characters if it needs to be extended.

The `chsize()` function ignores advisory locks that may have been set with the `fcntl()` function.

Returns:

- 0 Success.
- -1 An error occurred.

Errors:

- `EBADF` The `filedes` argument isn’t a valid file descriptor, or the file isn’t opened for writing.
chsize()

ENOSPC  There isn’t enough space left on the device to extend the file.
ENOSYS  The chsize() function isn’t implemented for the filesystem specified by filedes.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main( void )
{
    int filedes;
    filedes= open( "file", O_RDWR | O_CREAT,
                   S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
    if( filedes!= -1 ) {
        if( chsize( filedes, 32 * 1024L ) != 0 ) {
            printf( "Error extending file\n" );
        }
        close( filedes );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

close(), creat(), errno, ftruncate(), open()
Synopsis:

```c
#include <stdlib.h>

int clearenv( void );
```

Library:

```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `clearenv()` function clears the environment area; no environment variables are defined immediately after the `clearenv()` call.

Note that `clearenv()` clears the following environment variables, which may then affect the operation of other library functions such as `spawnp()`:

- PATH
- SHELL
- TERM
- TERMINFO
- LINES
- COLUMNS
- TZ

Returns:

0 Success.

-1 An error occurred (`errno` is set).
Errors:

ENOMEM Not enough memory to allocate a control structure.

Examples:

Clear the entire environment and set up a new TZ environment variable:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( clearenv() != 0 ) {
        puts( "Unable to clear the environment" );
        return EXIT_FAILURE;
    }

    setenv( "TZ", "EST5EDT", 0 );
    return EXIT_SUCCESS;
}
```

Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The clearenv() function manipulates the environment pointed to by the global environ variable.
See also:

environ, errno, exec(), execle(), execlp(), execvp(), execve(),
exejpe(), execvpe(), geteenv(), putenv(), searchenv(), seteenv(), spawn(),
spawnl(), spawne(), spawnp(), spawnlpe(), spawnlpe(), spawnlp(),
spawnv(), spawnve(), spawnvp(), spawnvpe(), system(), unseteenv()
clearerr()
Clear a stream's end-of-file and error flags

Synopsis:

```c
#include <stdio.h>

void clearerr( FILE *fp );
```

Arguments:

- `fp` The stream for which you want to clear the flags.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `clearerr()` function clears the end-of-file and error flags for the stream specified by `fp`.

These indicators are also cleared when the file is opened, or by an explicit call to `clearerr()` or `rewind()`.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    c = 'J';
    fp = fopen( "file", "w" );
    if( fp != NULL ) {
        fputc( c, fp );
        if( ferror( fp ) ) { /* if error */
            clearerr( fp ); /* clear the error */
            fputc( c, fp ); /* and retry it */
        }
    }
    fclose( fp );
```
clearerr()

    return EXIT_SUCCESS;
}

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

feof(), ferror(), fopen(), perror(), rewind()
**clock()**  
© 2007, QNX Software Systems GmbH & Co. KG.

*Return the number of clock ticks used by the program*

**Synopsis:**
```c
#include <time.h>

clock_t clock( void );
```

**Library:**
```
libc
```

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The *clock()* function returns the number of clock ticks of processor time used by the program since it started executing. You can convert the number of ticks into seconds by dividing by the value `CLOCKS_PER_SEC`.

In a multithreaded program, `clock()` returns the time used by *all* threads in the application; `clock()` returns the time since the program started, not the time since a specific thread started.

**Returns:**

The number of clock ticks, or `(clock_t) -1` if the number of ticks couldn’t be determined or exceeds the maximum value that the `clock_t` type can represent.

**Examples:**
```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

void compute( void )
{
    int i, j;
    double x;

    x = 0.0;
}  ```
for( i = 1; i <= 100; i++ ) {
    for( j = 1; j <= 100; j++ ) {
        x += sqrt( (double) i * j );
    }
}
printf( "%16.7f\n", x );
}

int main( void )
{
    clock_t start_time, end_time;
    start_time = clock();
    compute();
    end_time = clock();
    printf( "Execution time was %lu seconds\n",
            (long) ((end_time - start_time) / CLOCKS_PER_SEC) );
    return EXIT_SUCCESS;
}

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), ctime(), difftime(), gmtime(), localtime(),
localtime_r(), mktime(), strftime(), time(), tzset()
**clock_getcpuclockid()** © 2007, QNX Software Systems GmbH & Co. KG.

Return the clock ID of the CPU-time clock from a specified process

**Synopsis:**
```
#include <sys/types.h>
#include <time.h>

extern int clock_getcpuclockid(
    pid_t pid,
    clockid_t* clock_id
);
```

**Arguments:**
- `pid` The process ID for the process whose clock ID you want to get.
- `clock_id` A pointer to a `clockid_t` object where the function can store the clock ID.

**Library:**
`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `clock_getcpuclockid()` function returns the clock ID of the CPU-time clock of the process specified by `pid`. If the process described by `pid` exists and the calling process has permission, the clock ID of this clock is stored in `clock_id`.

If `pid` is zero, the clock ID of the CPU-time clock of the process marking the call is returned in `clock_id`.

A process always has permission to obtain the CPU-time clock ID of another process.

**Returns:**
Zero for success, or an error value.
**clock_getcpuclockid()**

**Errors:**

ESRCH  No process can be found corresponding to the specified `pid`.

**Classification:**

POSIX 1003.1 CPT

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`clock_getres()`, `clock_gettime()`, `ClockId()`, `clock_settime()`,
`pthread_getcpuclockid()`, `timer_create()`
clock_getres()  
Get the resolution of the clock

Synopsis:

#include <time.h>

int clock_getres( clockid_t clock_id,
                 struct timespec * res );

Arguments:

  clock_id     The ID of the clock whose resolution you want to get.
  res          A pointer to a timespec structure in which
               clock_getres() can store the resolution. The function
               sets the tv_sec member to 0, and the tv_nsec member to
               be the resolution of the clock, in nanoseconds.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The clock_getres() function gets the resolution of the clock specified
by clock_id and puts it into the buffer pointed to by res.

Returns:

  0      Success
  -1     An error occurred (errno is set).

Errors:

  EFAULT   A fault occurred trying to access the buffers provided.
  EINVAL   Invalid clock_id.
Examples:

```c
/*
 * This program prints out the clock resolution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    struct timespec res;

    if ( clock_getres( CLOCK_REALTIME, &res ) == -1 ) {
        perror( "clock get resolution" );
        return EXIT_FAILURE;
    }
    printf( "Resolution is \%ld micro seconds.\n",
            res.tv_nsec / 1000 );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`clock_gettime()`, `clock_settime()`, `ClockPeriod()`, `timespec`
**clock_gettime()**

*Get the current time of a clock*

**Synopsis:**

```c
#include <time.h>

int clock_gettime( clockid_t clock_id, 
                  struct timespec * tp );
```

**Arguments:**

- `clock_id` The ID of the clock whose time you want to get.
- `tp` A pointer to a `timespec` structure where `clock_gettime()` can store the time. This function sets the members as follows:
  - `tv_sec` — the number of seconds since 1970.
  - `tv_nsec` — the number of nanoseconds expired in the current second. This value increases by some multiple of nanoseconds, based on the system clock’s resolution.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `clock_gettime()` function gets the current time of the clock specified by `clock_id`, and puts it into the buffer pointed to by `tp`.

**Returns:**

- `0` Success.
- `-1` An error occurred (`errno` is set).
Errors:

EFAULT A fault occurred trying to access the buffers provided.
EINV AL Invalid clock_id.
ESRCH The process associated with this request doesn’t exist.

Examples:

/*
 * This program calculates the time required to
 * execute the program specified as its first argument.
 * The time is printed in seconds, on standard out.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#define BILLION 1000000000L;

int main( int argc, char** argv )
{
    struct timespec start, stop;
    double accum;

    if( clock_gettime( CLOCK_REALTIME, &start) == -1 ) {
        perror( "clock gettime" );
        return EXIT_FAILURE;
    }

    system( argv[1] );

    if( clock_gettime( CLOCK_REALTIME, &stop) == -1 ) {
        perror( "clock gettime" );
        return EXIT_FAILURE;
    }

    accum = ( (stop.tv_sec - start.tv_sec) 
            + (double)( stop.tv_nsec - start.tv_nsec )
          / (double)BILLION;
    printf( "%lf\n", accum );
    return EXIT_SUCCESS;
}
clock_gettime()  © 2007, QNX Software Systems GmbH & Co. KG.

Classification:

POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

clock_getres(), clock_settime(), errno, timespec
High resolution sleep with specifiable clock

Synopsis:
#include <time.h>

int clock_nanosleep( clockid_t clock_id,  
int flags,  
const struct timespec * rqtp,  
struct timespec * rmtp );

Arguments:

clock_id   The ID of the clock to use to measure the time. The possible clock types are:

CLOCK_MONOTONIC
A clock that always increases at a constant rate and can’t be adjusted.

CLOCK_SOFTTIME
Same as CLOCK_REALTIME, but if the CPU is in powerdown mode, the clock stops running.

CLOCK_REALTIME
A clock that maintains the system time.

The clock_nanosleep() function fails if the clock_id argument refers to the CPU-time clock of the calling thread.

flags   Flags that specify when the current thread is to be suspended from execution:

• when the time interval specified by the rqtp argument has elapsed (TIMER_ABSTIME is not set).

• when the time value of the clock specified by clock_id reaches the absolute time specified by the rqtp argument (TIMER_ABSTIME is set).

If, at the time of the call, the time value specified by rqtp is less than or equal to the time value of the specified clock, then clock_nanosleep() returns immediately, and the calling process isn’t suspended.
\textit{clock\_nanosleep()}  

- when a signal is delivered to the calling thread, and the signal’s action is to invoke a signal-catching function or terminate the process.

Calling \textit{clock\_nanosleep()} with \texttt{TIMER\_ABSTIME} not set, and \texttt{clock\_id} set to \texttt{CLOCK\_REALTIME} is the equivalent to calling \textit{nanosleep()} with the same \texttt{rqtp} and \texttt{rmtt} arguments.

**rqtp**  
A pointer to a \texttt{timespec} structure that specifies the time interval between the requested time and the time actually slept.

**rmtt**  
NULL, or a pointer to a \texttt{timespec} in which the function can store the amount of time remaining in an interval.

For the relative \textit{clock\_nanosleep()} function, if \texttt{rmtt} isn’t NULL, the \texttt{timespec} structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If it’s NULL, the remaining time isn’t returned.

The absolute \textit{clock\_nanosleep()} function has no effect on the structure referenced by \texttt{rmtt}.

**Library:**

\texttt{libc}

Use the \texttt{-lc} option to \texttt{qcc} to link against this library. This library is usually included automatically.

**Description:**

The \textit{clock\_nanosleep()} function suspends the current thread from execution until:

- If \texttt{TIMER\_ABSTIME} is set, the time value of the clock specified by \texttt{clock\_id} reaches the absolute time specified by the \texttt{rqtp} argument.

Or:
If TIMER_ABSTIME is not set, the time interval specified by the `rqtp` argument has elapsed.

Or:

A signal is delivered to the calling thread, and the signal’s action is to invoke a signal-catching function or terminate the process.

The `nanosleep()` function always uses CLOCK_REALTIME.

The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of scheduling and other system activity. Except for the case of being interrupted by a signal, the suspension time for:

- the relative `clock_nanosleep()` function (TIMER_ABSTIME not set) — isn’t less than the time interval specified by `rqtp`, as measured by the corresponding clock
- the absolute `clock_nanosleep()` function (TIMER_ABSTIME set) — is in effect at least until the value of the corresponding clock reaches the absolute time specified by `rqtp`, except for the case of being interrupted by a signal.

Using the `clock_nanosleep()` function has no effect on the action or blockage of any signal.

**Returns:**

Zero if the requested time has elapsed, or a corresponding error value if `clock_nanosleep()` has been interrupted by a signal, or fails.

**Errors:**

EINTR The call was interrupted by a signal.

EINVAL The `rqtp` argument specified a nanosecond value less than zero or greater than or equal to 1000 million; or TIMER_ABSTIME is specified in `flags` and the `rqtp` argument is outside the range for the clock specified by `clock_id`; or the `clock_id` argument doesn’t specify
clock_nanosleep()  © 2007, QNX Software Systems GmbH & Co. KG.

a known clock, or specifies the CPU-time clock of the calling thread.

ENOTSUP The clock_id argument specifies a clock for which clock_nanosleep() isn’t supported, such as a CPU-time clock.

Classification:

POSIX 1003.1 CS

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

clock_settime(), nanosleep(), sleep(), timespec


**clock_settime()**

*Set a clock*

**Synopsis:**

```c
#include <time.h>

int clock_settime( clockid_t id,
                   const struct timespec * tp );
```

**Arguments:**

- `id`  The clock ID, CLOCK_REALTIME or CLOCK_MONOTONIC, that maintains the system time, or the clock ID that’s returned by ClockId().

- `tp`  A pointer to a `timespec` structure containing at least the following members:
  - `tv_sec` — the number of seconds since 1970.
  - `tv_nsec` — the number of nanoseconds in the current second. This value increases by some multiple of nanoseconds, based on the system clock’s resolution.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `clock_settime()` function sets the clock specified by `id` to the time specified in the buffer pointed to by `tp`. 
Be careful if you set the date during the period that a time zone is switching from daylight saving time (DST) to standard time. When a time zone changes to standard time, the local time goes back one hour (for example, 2:00 a.m. becomes 1:00 a.m.). The local time during this hour is ambiguous (e.g. 1:14 a.m. occurs twice in the morning that the time zone switches to standard time). To avoid problems, use UTC time to set the date in this period.

- You can’t set the time when the id is CLOCK_MONOTONIC.
- You need to have superuser privileges to set the clock.

Returns:

0 Success

-1 An error occurred (errno is set).

Errors:

EINVAL Invalid id or the number of nanoseconds specified by the tv_nsec is less than zero or greater than or equal to 1000 million.

EPERM You don’t have sufficient privilege to change the time.

Examples:

/* This program sets the clock forward 1 day. */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

int main( void )
{
    struct timespec stime;

    if( clock_gettime( CLOCK_REALTIME, &stime) == -1 ) {
        perror( "getclock" );
    }
    
}
clock_settime()

    return EXIT_FAILURE;
}

stime.tv_sec += (60*60)*24L; /* Add one day */
stime.tv_nsec = 0;
if( clock_settime( CLOCK_REALTIME, &stime) == -1 ) {
    perror( "setclock" );
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;

Classification:

    POSIX 1003.1 TMR

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

clock_getres(), clock_gettime(), errno, timespec
Adjust the time of a clock

Synopsis:

```c
#include <sys/neutrino.h>

int ClockAdjust( clockid_t id,
                 const struct _clockadjust * new,
                 struct _clockadjust * old );

int ClockAdjust_r( clockid_t id,
                   const struct _clockadjust * new,
                   struct _clockadjust * old );
```

Arguments:

- **id**
  The ID of the clock you want to adjust. This must be `CLOCK_REALTIME`; this clock maintains the system time.

- **new**
  NULL or a pointer to a `_clockadjust` structure that specifies how to adjust the clock. Any previous adjustment is replaced.
  
  The `_clockadjust` structure contains at least the following members:
  
  - `long tick_nsec_inc` — the adjustment to be made on each clock tick, in nanoseconds.
  - `unsigned long tick_count` — the number of clock ticks over which to apply the adjustment.

- **old**
  If not NULL, a pointer to a `_clockadjust` structure where the function can store the current adjustment (before being changed by a non-NULL `new`).

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

These kernel calls let you gradually adjust the time of the clock specified by id. You can use these functions to speed up or slow down the system clock to synchronize it with another time source – without causing major discontinuities in the time flow.

The ClockAdjust() and ClockAdjust_r() functions are identical except in the way they indicate errors. See the Returns section for details.

The total time adjustment, in nanoseconds, is:

\[(\text{new->tick\_count} \times \text{new->tick\_nsec\_inc})\]

If the current clock is ahead of the desired time, you can specify a negative tick\_nsec\_inc to slow down the clock. This is preferable to setting the time backwards with the ClockTime() kernel call, since some programs may malfunction if time goes backwards.

Picking small values for tick\_nsec\_inc and large values for tick\_count adjusts the time slowly, while the opposite approach adjusts it rapidly. As a rule of thumb, don’t try to set a tick\_nsec\_inc that exceeds the basic clock tick as set by the ClockPeriod() kernel call. This would change the clock rate by more than 100% and if the adjustment is negative, it could make the clock go backwards.

You can cancel any adjustment in progress by setting tick\_count and tick\_nsec\_inc to 0.

Superuser privileges are required to adjust the clock.

Blocking states:

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

- ClockAdjust() If an error occurs, the function returns -1 and sets errno. Any other value returned indicates success.
ClockAdjust(), ClockAdjust_r()

ClockAdjust_r() EOK is returned on success. This function does NOT set errno. If an error occurs, the function may return any value in the Errors section.

Errors:

EFAULT A fault occurred when the kernel tried to access the buffers provided.
EINV AL The clock id isn’t valid.
EPERM The process tried to adjust the time without having superuser capabilities.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

ClockPeriod(), ClockTime()
ClockCycles()

Get the number of clock cycles

Synopsis:

```c
#include <sys/neutrino.h>
#include <inttypes.h>

uint64_t ClockCycles( void );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The ClockCycles() kernel call returns the current value of a free-running 64-bit cycle counter. This is implemented on each processor as a high-performance mechanism for timing short intervals.

Several CPU architectures have an instruction that reads such a free-running counter (e.g. x86 has the RDTSC instruction). For processors that don’t implement such an instruction in hardware (e.g. a 386), the kernel emulates one. This provides a lower time resolution than if the instruction is provided (838.095345 nanoseconds on an IBM PC-compatible system).

In all cases, the `SYSPAGE_ENTRY(qtime)->cycles_per_sec` field gives the number of ClockCycles() increments in one second.

Symmetric MultiProcessing systems

This function, depending on the CPU architecture, returns a value from a register that’s unique to each CPU in an SMP system — for instance, the TSC (Time Stamp Counter) on an x86. These registers aren’t synchronized between the CPUs. So if you call ClockCycles(), and then the thread migrates to another CPU and you call ClockCycles() again, you can’t subtract the two values to get a meaningful time duration.
If you wish to use `ClockCycles()` on an SMP machine, you must use the following call to “lock” the thread to a single CPU:

`ThreadCtl(_NTO_TCTL_RUNMASK, ...)`

### Blocking states:

This call doesn’t block.

---

**CAUTION:**

Be careful about wrapping of the cycle counter. Use the following to calculate how many seconds before the cycle counter wraps:

\[
\frac{\sim (\text{uint64}_t)0}{\text{SYSPAGE\_ENTRY}(qtime)} \to \text{cycles\_per\_sec}
\]

### Examples:

See `SYSPAGE\_ENTRY()`.

### Classification:

QNX Neutrino

#### Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`SYSPAGE\_ENTRY()`, `ThreadCtl()`
**ClockId(), ClockId_r()**

Get a clock ID for a given process and thread

**Synopsis:**

```c
#include <sys/neutrino.h>
#include <inttypes.h>

extern int ClockId( pid_t pid,
                    int tid );

extern int ClockId_r( pid_t pid,
                      int tid );
```

**Arguments:**

- **pid** The ID of the process that you want to calculate the execution time for. If this argument is zero, the ID of the process making the call is assumed.

- **tid** The ID of the thread that you want to calculate the execution time for, or 0 to get the execution time for the process as a whole.

**Library:**

```
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `ClockId()` and `ClockId_r()` kernel calls return an integer that you can pass as a `clockid_t` to `ClockTime()`. When you pass this clock ID to `ClockTime()`, the function returns (in the location pointed to by `old`) the number of nanoseconds that the specified thread of the specified process has executed.

The `ClockId()` and `ClockId_r()` functions are identical except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling `clock_getcpuclockid()` or `pthread_getcpuclockid()`.

If the `tid` is zero, the number of nanoseconds that the process as a whole has executed is returned. On an SMP box, this number may exceed the realtime number of nanoseconds that have elapsed because multiple threads in the process can run on several CPUs at the same time.

**Blocking states:**

This call doesn’t block.

**Returns:**

- `ClockId()` An integer that can be passed to `ClockTime()`. If an error occurs, the function returns -1 and sets `errno`.
- `ClockId_r()` An integer that can be passed to `ClockTime()`. This function does **NOT** set `errno`. If an error occurs, the function returns the negative of a value from the Errors section.

**Errors:**

- ESRCH The `pid` and/or `tid` don’t exist.

**Examples:**

Here’s how you can determine how busy a system is:

```c
id = ClockId(1, 1);
for(;; ) {
    ClockTime(id, NULL, &start);
    sleep(1);
    ClockTime(id, NULL, &stop);
    printf("load = %f\n", (1000000000.0 - (stop-start)) / 10000000.0);
}
```
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

ClockId(), clock_gettime(), pthread_getcpuclockid()
**ClockPeriod(), ClockPeriod_r()**


Get or set a clock period

**Synopsis:**

```c
#include <sys/neutrino.h>

int ClockPeriod( clockid_t id,
                 const struct _clockperiod * new,
                 struct _clockperiod * old,
                 int reserved );

int ClockPeriod_r( clockid_t id,
                   const struct _clockperiod * new,
                   struct _clockperiod * old,
                   int reserved );
```

**Arguments:**

- **id**
  The clock ID of the clock. This must be CLOCK_REALTIME, which is the ID of the clock that maintains the system time.

- **new**
  NULL, or a pointer to a _clockperiod structure that contains the period to set the clock to. This structure contains at least the following members:
  - **unsigned long nsec** — the period of the clock, in nanoseconds.
  - **long fract** — reserved for future fractional nanoseconds. Set this member to zero.

- **old**
  NULL, or a pointer to a _clockperiod structure where the function can store the current period (before being changed by a non-NULL new).

- **reserved**
  Set this argument to 0.

**Library:**

- **libc**
  Use the `-l c` option to qcc to link against this library. This library is usually included automatically.
Description:

You can use the ClockPeriod() and ClockPeriod_r() kernel calls to get or set the clock period of the clock.

If you want to get the clock period, consider calling clock_getres() instead of using these kernel calls directly.

These functions are identical except in the way they indicate errors. See the Returns section for details.

- You need to have superuser privileges to set the clock period.
- If you’re using adaptive partitioning, and you change the tick size of the system at runtime, do so before defining the adaptive partitioning scheduler’s window size. That’s because Neutrino converts the window size from milliseconds to clock ticks for internal use. For more information, see the Adaptive Partitioning User’s Guide.

All the timer_*() calls operate with an accuracy no better than the clock period. Every moment within the Neutrino microkernel is referred to as a tick. A tick’s initial length is determined by the clock rate of your processor:

<table>
<thead>
<tr>
<th>CPU clock speed</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥ 40MHz</td>
<td>1 millisecond</td>
</tr>
<tr>
<td>&lt; 40MHz</td>
<td>10 milliseconds</td>
</tr>
</tbody>
</table>

Since a very small ticksize imposes an interrupt load on the system, and can consume all available processor cycles, the kernel call limits how small a period can be specified. The lowest clock period that can currently be set on any machine is 10 microseconds.

If an attempt is made to set a value that the kernel believes to be unsafe, the call fails with an EINVAL. The timeslice rate (for
“round-robin” and “other” scheduling policies) is always four times the clock period (this isn’t changeable).

Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

ClockPeriod()

If an error occurs, this function returns -1 is and sets errno. Any other value returned indicates success.

ClockPeriod_r()

EOK is returned on success. This function does NOT set errno. If an error occurs, the function can return any value in the Errors section.

Errors:

EFAULT A fault occurred when the kernel tried to access the buffers provided.

EINVAL Invalid clock ID. A period was set which wasn’t in a range considered safe.

EPERM The process tried to change the period of the clock without having superuser capabilities.

Classification:

QNX Neutrino

Safety

Cancellation point  No

continued . . .
### ClockPeriod(), ClockPeriod_r()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`clock_getres()`, `ClockAdjust()`
ClockTime(), ClockTime_r()

Get or set a clock

Synopsis:

```c
#include <sys/neutrino.h>

int ClockTime( clockid_t id,
               const uint64_t * new,
               uint64_t * old);

int ClockTime_r( clockid_t id,
                const uint64_t * new,
                uint64_t * old);
```

Arguments:

- `id`  
The clock ID. This must be CLOCK_REALTIME or CLOCK_MONOTONIC, which is the ID of the clock that maintains the system time, or the clock ID that’s returned by ClockId().

- `new`  
NULL, or a pointer to the absolute time, in nanoseconds, to set the clock to. This is ignored when `id` is CLOCK_MONOTONIC.

- `old`  
NULL, or a pointer to a location where the function can store the current time (before being changed by a non-NULL `new`).

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use these kernel calls to get or set the system clock specified by `id`. The clock ID, CLOCK_REALTIME or CLOCK_MONOTONIC, maintains the system time.

The `ClockTime()` and `ClockTime_r()` functions are identical except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling `clock_gettime()` or `clock_settime()`.

If `new` isn’t NULL, then it contains the absolute time, in nanoseconds, to set the system clock to. This affects the software clock maintained by the system. It doesn’t change any underlying hardware clock that maintains the time when the system’s power is turned off.

You can’t set the time when the `id` is `CLOCK_MONOTONIC`.

Once set, the system time increments by some number of nanoseconds, based on the resolution of the system clock. You can query or change this resolution by using the `ClockPeriod()` kernel call.

You need to have superuser privileges to set the clock.

**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

- `ClockTime()` If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

- `ClockTime_r()` `EOK` is returned on success. This function does **NOT** set `errno`. If an error occurs, the function returns a value in the Errors section.

**Errors:**

- `EFAULT` A fault occurred when the kernel tried to access the buffers provided.
ClockTime(), ClockTime_r()  © 2007, QNX Software Systems GmbH & Co. KG.

EINVAL  The clock ID isn’t CLOCK_REALTIME or CLOCK_MONOTONIC.

EPERM  The process tried to change the time without having superuser capabilities.

ESRCH  The process associated with this request doesn’t exist.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*clock_gettime(), clock_settime(), ClockAdjust(), ClockPeriod()*
close()
Close a file

Synopsis:

```c
#include <unistd.h>

int close( int filedes );
```

Arguments:

`filedes` The file descriptor of the file you want to close. This can be a file descriptor returned by a successful call to `accept()`, `creat()`, `dup()`, `dup2()`, `fcntl()`, `modem_open()`, `open()`, `shm_open()`, `socket()` or `sopen()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `close()` function closes the file specified by the given file descriptor.

Returns:

Zero for success, or -1 if an error occurs (errno is set).

Errors:

- **EBADF**: Invalid file descriptor `filedes`.
- **EINTR**: The `close()` call was interrupted by a signal.
- **EIO**: An I/O error occurred while updating the directory information.
- **ENOSPC**: A previous buffered write call has failed.
- **ENOSYS**: The `close()` function isn’t implemented for the filesystem specified by `filedes`.

September 10, 2007 QNX Neutrino Functions and Macros 361
close()

Examples:

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes;

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        /* process file */
        close( filedes );
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

accept(), creat(), dup(), dup2(), errno, fcntl(), modem_open(), open(), shm_open(), socket(), sopen()
Synopsis:

```c
#include <dirent.h>

int closedir( DIR * dirp );
```

Arguments:

- `dirp` A directory pointer for the directory you want to close.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `closedir()` function closes the directory specified by `dirp`, and frees the memory allocated by `opendir()`.

The result of using a directory stream after calling one of the `exec*()` or `spawn*()` family of functions is undefined. After a call to the `fork()` function, either the parent or the child (but not both) may continue processing the directory stream using the `readdir()` and `rewinddir()` functions. If both the parent and child processes use these functions, the result is undefined. Either or both processes may call the `closedir()` function.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).
Errors:

EBADF   The dirp argument doesn’t refer to an open directory stream.

EINTR   The closedir() call was interrupted by a signal.

Examples:

Get a list of files contained in the directory /home/kenny:

```c
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

int main( void )
{
    DIR *dirp;
    struct dirent *direntp;

    dirp = opendir( "/home/kenny" );
    if( dirp != NULL ) {
        for(;;) {
           direntp = readdir( dirp );
            if( direntp == NULL ) {
                break;
            }
            printf( "%s\n", direntp->d_name );
        }
        closedir( dirp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

POSIX 1003.1
closedir()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

erno, opendir(), readdir(), readdir_r(), rewinddir(), seekdir(), telldir()
closelog()

Close the system log

Synopsis:

```c
#include <syslog.h>

void closelog( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `closelog()` function closes the connection to `syslogd`.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`openlog()`, `setlogmask()`, `syslog()`, `vsyslog()`, `logger`, `syslogd` in the `Utilities Reference`
**Synopsis:**

```c
#include <process.h>

int _cmdfd( void );
```

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

This function returns a file descriptor for the executable file.

**Returns:**

A file descriptor for the executable file.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`_cmdname()`, `__progname`
Synopsis:

```c
#include <process.h>

char * _cmdname( char * buff );
```

Arguments:

- `buff`: NULL, or a pointer to a buffer in which the function can store the path. To determine the size required for the buffer, call `fpathconf()` or `pathconf()` with an argument of `_PC_PATH_MAX`, then add 1 for the terminating null character.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_cmdname()` function determines the full path that the current process was invoked from. If `buff` isn’t NULL, `_cmdname()` copies the path into the buffer that `buff` points to.

Returns:

A pointer to the pathname used to load the process, or NULL if an error occurred.

Don’t change the string that the returned value points to if you passed NULL for the `buff` parameter.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
```
#include <process.h>

int main( void )
{
    size_t maximum_path;
    char *buff;

    maximum_path = (size_t) pathconf( "/", _PC_PATH_MAX );
    buff = (char*) malloc( maximum_path );

    if( _cmdname( buff ) ) {
        printf( "I'm "%s".\n", buff );
    } else {
        perror( "_cmdname() failed" );
        free (buff);
        return EXIT_FAILURE;
    }

    free (buff);
    return EXIT_SUCCESS;
}

If this code is compiled into an executable named foo:

    # ls -F /home/xyzzy/bin/foo
    foo*
    # /home/xyzzy/bin/foo
    I'm "/home/xyzzy/bin/foo".

Classification:

QNX 4

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

basename(), cmdfd(), __proname
Synopsis:

```
#include <unistd.h>

size_t confstr( int name,
                char * buf,
                size_t len );
```

Arguments:

- **name**: The system variable to query; see below.
- **buf**: A pointer to a buffer in which the function can store the value of the system variable.
- **len**: The length of the buffer, in bytes.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `confstr()` functions lets applications get or set configuration-defined string values. This is similar to the `sysconf()` function, but you use it to get string values, rather than numeric values. By default, the function queries and returns values in the system.

The `name` argument represents the system variable to query. The values are defined in `<confname.h>`; at least the following `name` values are valid:

```
_CS_ARCHITECTURE
```

The name of the instruction set architecture for this node’s CPU(s).
`confstr()`

_Available configurations are:_

- `_CS_DOMAIN_` The domain name.
- `_CS_HOSTNAME_` The name of this node in the network.

A hostname can consist only of letters, numbers, and hyphens, and must not start or end with a hyphen. For more information, see RFC 952.

- `_CS_HW_PROVIDER_` The name of the hardware manufacturer.
- `_CS_HW_SERIAL_` Serial number associated with the hardware.
- `_CS_LIBPATH_` A value similar to the `LD_LIBRARY_PATH` environment variable that finds all standard libraries.
- `_CS_MACHINE_` This node’s hardware type.
- `_CS_PATH_` A value similar to the `PATH` environment variable that finds all standard utilities.
- `_CS_RELEASE_` The current OS release level.
- `_CS_RESOLVE_` The contents of the `resolv.conf` file, excluding the domain name.
- `_CS_SRPC_DOMAIN_` The secure RPC domain.
- `_CS_SYSNAME_` The operating system name.
- `_CS_TIMEZONE_` Time zone string (TZ style)
- `_CS_VERSION_` The current OS version number.

The configuration-defined value is returned in the buffer pointed to by `buf`, and will be ≤ `len` bytes long, including the terminating NULL.
To find out the length of a configuration-defined value, call `confstr()` with `buf` set to NULL and `len` set to 0.

To set a configuration value:
- OR your value to be defined (i.e. `_CS_HOSTNAME`) to `_CS_SET`
- put this value in a NULL-terminated string
- Set the value of `len` to 0

Returns:
A nonzero value (if a “get” is done, the value is the length of the configuration-defined value), or 0 if an error occurs (`errno` is set).

You can compare the `confstr()` return value against `len` to see if the configuration-defined value was truncated when retrieving a value, (this can’t be done when setting a value).

Errors:
EINVAL The `name` argument isn’t a valid configuration-defined value.

Examples:
Print information similar to that returned by the `uname()` function:

```c
#include <unistd.h>
#include <stdio.h>
#include <limits.h>

#define BUFF_SIZE (256 + 1)

int main( void )
{
    char buff[BUFF_SIZE];

    if( confstr( _CS_SYSNAME, buff, BUFF_SIZE ) > 0 ) {
        printf( "System name: %s\n", buff );
    }

    if( confstr( _CS_HOSTNAME, buff, BUFF_SIZE ) > 0 ) {
        printf( "Host name: %s\n", buff );
    }
```
if( confstr( _CS_RELEASE, buff, BUFF_SIZE ) > 0 ) {
    printf( "Release: %s\n", buff );
}

if( confstr( _CS_VERSION, buff, BUFF_SIZE ) > 0 ) {
    printf( "Version: %s\n", buff );
}

if( confstr( _CS_MACHINE, buff, BUFF_SIZE ) > 0 ) {
    printf( "Machine: %s\n", buff );
}

if( confstr( _CS_SET | _CS_HOSTNAME, "myhostname", 0 ) != 0 ) {
    printf( "Hostname set to: %s\n", "myhostname" );
}
return 0;

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The confstr() function is part of a draft standard; its interface and/or behavior may change in the future.
See also:

pathconf(), sysconf()

getconf, setconf in the Utilities Reference

“Configuration strings” in the Configuring Your Environment chapter of the Neutrino User’s Guide
connect()

Initiate a connection on a socket

Synopsis:
```
#include <sys/types.h>
#include <sys/socket.h>

int connect( int s,
    const struct sockaddr * name,
    socklen_t namelen );
```

Arguments:
- `s` The descriptor of the socket on which to initiate the connection.
- `name` The name of the socket to connect to for a SOCK_STREAM connection.
- `namelen` The length of the `name`, in bytes.

Library:
```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

Description:
The `connect()` function establishes the connection according to the socket type specified by `s`:

- **SOCK_DGRAM**
  Specifies the peer that the socket is to be associated with. This address is the one that datagrams are to be sent to, and the only one that datagrams are to be received from.

- **SOCK_STREAM**
  This call attempts to make a connection to another socket. The other socket is specified by `name`, which is an address in the communications space of that socket. Each communications space interprets `name` in its own way.
Stream sockets may successfully connect only once, whereas datagram sockets may use `connect()` multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

**Returns:**

0 Success.

-1 An error occurred (`errno` is set).

**Errors:**

- `EADDRINUSE` The address is already in use.
- `EADDRNOTAVAIL` The specified address isn’t available on this machine.
- `EAFNOSUPPORT` Addresses in the specified address family cannot be used with this socket.
- `EALREADY` The socket is nonblocking; a previous connection attempt hasn’t yet been completed.
- `EBADF` Invalid descriptor `s`.
- `ECONNABORTED` The `connect()` was terminated under software control.
- `ECONNREFUSED` The attempt to connect was forcefully rejected.
- `EFAULT` The `name` parameter specifies an area outside the process address space.
- `EHOSTUNREACH` No route to host; the host system can’t be reached.
**connect()**

EINPROGRESS  The socket is nonblocking; the connection can’t be completed immediately. It’s possible to do a `select()` for completion by selecting the socket for writing.

EISCONN   The socket is already connected.

ENETUNREACH  The network isn’t reachable from this host.

ETIMEDOUT  The attempt to establish a connection timed out; no connection was made.

Protocols such as TCP do not allow further connection requests on a socket after an ECONNREFUSED error. In such a situation, the socket must be closed and a new one created before a subsequent attempt for connection is made.

**Classification:**

POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

ICMP, IP, TCP, and UDP protocols

`accept()`, `bind()`, `getsockname()`, `nbaconnect()`, `select()`, `socket()`
ConnectAttach(),
ConnectAttach_r()

Establish a connection between a process and a channel

Synopsis:

```c
#include <sys/neutrino.h>

int ConnectAttach( uint32_t nd,
                    pid_t pid,
                    int chid,
                    unsigned index,
                    int flags);

int ConnectAttach_r( uint32_t nd,
                     pid_t pid,
                     int chid,
                     unsigned index,
                     int flags);
```

Arguments:

- **nd** The node descriptor of the node on which the process that owns the channel is running; see “Node descriptors,” below.
- **pid** The process ID of the owner of the channel. If *pid* is zero, the calling process is assumed.
- **chid** The channel ID, returned by `ChannelCreate()`, of the channel to connect to the process.
- **index** The lowest acceptable connection ID.
Treating a connection as a file descriptor can lead to unexpected behavior. Therefore, you should OR \_NTO\_SIDE\_CHANNEL into \textit{index} when you create a connection. If you do this, the connection ID is returned from a different space than file descriptors; the ID is greater than any valid file descriptor.

Once created there’s no difference in the use of the messaging primitives on this ID. The C library creates connections at various times without \_NTO\_SIDE\_CHANNEL (e.g. during \textit{open}()), however, it’s unlikely that any applications would want to call it this way.

\begin{itemize}
\item[\textbf{flags}] If flags contains \_NTO\_COF\_CLOEXEC, the connection is closed when your process calls an \textit{exec*}() function to start a new process.
\end{itemize}

**Library:**

\texttt{libc}

Use the -l c option to \texttt{qcc} to link against this library. This library is usually included automatically.

**Description:**

The \textit{ConnectAttach()} and \textit{ConnectAttach \_r()} kernel calls establish a connection between the calling process and the channel specified by \textit{chid} owned by the process specified by \textit{pid} on the node specified by \textit{nd}. Any function that passes a node descriptor can use either the value 0 or the constant ND\_LOCAL\_NODE to refer to the local node.

These functions are identical except in the way they indicate errors. See the Returns section for details.

The return value is a connection ID, which is a small \texttt{int} representing the connection. The system returns the first available connection ID starting at the value specified by the \textit{index} argument. Any thread in the calling process can use either \textit{MsgSendv()} to send messages or \textit{MsgSendPulse()} to send pulses over the connection. The connection ID is used directly as a POSIX file descriptor (\textit{fd}) when
communicating with I/O Resource managers such as a filesystem manager.

If you don’t OR _NTO_SIDE_CHANNEL into index, this behavior might result:

- If file descriptor 0 is in use, file descriptor 1 isn’t in use, and you call ConnectAttach() with 0 specified for index, a connection ID of 1 is returned.
  File descriptor 1 (i.e. connection ID 1) is used as stdout, which is what printf() writes to. If your process makes any calls to printf(), NULL-terminated character strings are sent to the channel that you’ve connected to. Similar situations can happen with connection IDs 0 (stdin) and 2 (stderr).

- Depending on how a child process is created, it may inherit the parent’s file descriptors.
  Since connections are treated like file descriptors, a connection created by the parent without _NTO_SIDE_CHANNEL in index and without _NTO_COF_CLOEXEC in flags, causes a child process to inherit that connection during process creation. This inheritance is done during process creation by duplicating file descriptors.
  During duplication, an _IO_DUP message (with 0x115) as the first 2 bytes) is sent to the receiver on the other side of the connection. The receiver won’t be expecting this message.

If index has _NTO_SIDE_CHANNEL set, the index is ignored and the connection ID returned is the first available index in the _NTO_SIDE_CHANNEL space.

If a process creates multiple connections to the same channel, the system maintains a link count and shares internal kernel object resources for efficiency.

Connections are owned by the process and may be used simultaneously by any thread in the process. You can detach a connection by calling ConnectDetach(). If any threads are blocked on the channel (via MsgSendv()) at the time the connection is detached, the send fails and returns with an error.
Connections and connection IDs persist until you call ConnectDetach(), even if the other process dies.

The connection is strictly local (i.e. it doesn’t resolve across the network) and is resolved on the first use of the connection ID.

Blocking states
These calls don’t block.

Node descriptors
The \( nd \) (node descriptor) is a temporary numeric description of a remote node. For more information, see the Qnet Networking chapter of the System Architecture guide.

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare two ( nd ) objects</td>
<td>( ND_NODE_CMP() )</td>
</tr>
<tr>
<td>Convert a ( nd ) to text</td>
<td>netmgr_ndtostr()</td>
</tr>
<tr>
<td>Convert text to a ( nd )</td>
<td>netmgr_strtond()</td>
</tr>
</tbody>
</table>

Returns:
The only difference between these functions is the way they indicate errors:

\( \text{ConnectAttach()} \)
A connection ID that’s used by the message primitives. If an error occurs, the function returns \(-1\) and sets \( \text{errno} \).

\( \text{ConnectAttach}_r() \)
A connection ID that’s used by the message primitives. This function does NOT set \( \text{errno} \). If an error occurs, the function returns the negative of a value from the Errors section.
ConnectAttach(),
ConnectAttach_r()

Errors:

EAGAIN All kernel connection objects are in use.
ESRCH The node indicated by \textit{nd}, the process indicated by \textit{pid},
or the channel indicated by \textit{chid} doesn’t exist.

Classification:

\textbf{QNX Neutrino}

\begin{center}
\begin{tabular}{ll}
\textbf{Safety} & \\
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}
\end{center}

See also:

ChannelCreate(), ConnectDetach(), execl(), execle(), execlpe(),
exclpe(), execv(), execve(), execvp(), execvep(), MsgSendPulse(),
MsgSendv(), netmgr\_remote\_nd()
ConnectClientInfo(), ConnectClientInfo_r()

Synopsis:

```
#include <sys/neutrino.h>

int ConnectClientInfo(int scoid, struct _client_info *info, int ngroups);

int ConnectClientInfo_r(int scoid, struct _client_info *info, int ngroups);
```

Arguments:

- `scoid` A server connection ID that identifies the client process that you want to get information about. This client is typically a process that's made a connection to the server to try to access a resource. You can get it from the `_msg_info` argument to `MsgReceivev()` or `MsgInfo()`.

- `info` A pointer to a `_client_info` structure that the function can fill with information about the client. For more information, see below.

- `ngroups` The size of the caller's `grouplist` in the credential part of the `_client_info` structure. If you make it smaller than `NGROUPS_MAX`, you might get information only about a subset of the groups.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These calls get information about a client connection identified by `scoid`, and store it in the buffer that `info` points to.
The `ConnectClientInfo()` and `ConnectClientInfo_r()` functions are identical except in the way they indicate errors. See the Returns section for details. A server uses these functions to determine whether or not a client has permission to access a resource. For example, in a resource manager, it would be called on an `open()` connection request.

**_client_info structure**

The `_client_info` structure has at least the following members:

- `uint32_t nd`: The client’s node ID.
- `pid_t pid`: The client’s process ID.
- `struct _cred_info cred`: The user and group ID credentials; see below.

**uint32_t nd**

The `nd` (node descriptor) is a temporary numeric description of a remote node. For more information, see the Qnet Networking chapter of the System Architecture guide.

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare two <code>nd</code> objects</td>
<td><code>ND_NODE_CMP()</code></td>
</tr>
<tr>
<td>Convert a <code>nd</code> to text</td>
<td><code>netmgr_ndtostr()</code></td>
</tr>
<tr>
<td>Convert text to a <code>nd</code></td>
<td><code>netmgr_strtond()</code></td>
</tr>
</tbody>
</table>

**_cred_info structure**

The `cred` member of the `_client_info` is a `_cred_info` structure that includes at least the following members:

- `uid_t ruid`: The real user ID of the sending process.
uid_t euid    The effective user ID of the sending process.
uid_t suid    The saved user ID of the sending process.
gid_t rgid    The real group ID of the sending process.
gid_t egid    The effective group ID of the sending process.
gid_t sgid    The saved group ID of the sending process.
uint32_t ngroups
              The number of groups actually stored in grouplist.
gid_t grouplist[NGROUPS_MAX]
              The supplementary group IDs of the sending process.

The ngroups argument to ConnectClientInfo() indicates the size of the
grouplist array. If the group array size is zero, the ngroups member of
the _cred_info is set to the number of groups available.

Returns:

The only difference between these functions is the way they indicate
errors:

ConnectClientInfo()

If an error occurs, the function returns -1 and sets errno. Any
other value returned indicates success.

ConnectClientInfo_r()

EOK is returned on success. This function does NOT set errno.
If an error occurs, the function can return any value in the
Errors section.

Errors:

EFAULT   A fault occurred when the kernel tried to access the
           buffers provided.
EINVAL    Process doesn’t have a connection scoid.
ConnectClientInfo(), ConnectClientInfo_r()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ConnectServerInfo(), _msg_info, MsgInfo(), MsgReceivev(), ND_NODE_CMP(), netmgr_ndtostr(), netmgr_remote_nd(), netmgr_strtond()
ConnectDetach(), ConnectDetach_r()

Break a connection between a process and a channel

Synopsis:

```c
#include <sys/neutrino.h>

int ConnectDetach( int coid );

int ConnectDetach_r( int coid );
```

Arguments:

- `coid` The connection ID of the connection you want to break.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ConnectDetach()` and `ConnectDetach_r()` kernel calls detach the connection specified by the `coid` argument. If any threads are blocked on the connection (`MsgSendv()`) at the time the connection is detached, the send fails and returns with an error.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

- `ConnectDetach()`
  
  If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.
**ConnectDetach**

EOK is returned on success. This function does **NOT** set `errno`. If an error occurs, the function returns a value in the Errors section.

**Errors:**

- **EINVAL**  The connection specified by `coid` doesn’t exist.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*ConnectAttach*, *MsgSendv*
Modify the flags associated with a connection

Synopsis:
```c
#include <sys/neutrino.h>

int ConnectFlags( pid_t pid,
                  int coid,
                  unsigned mask,
                  unsigned bits );

int ConnectFlags_r( pid_t pid,
                    int coid,
                    unsigned mask,
                    unsigned bits );
```

Arguments:
- **pid** The ID of the process that the connection ID belongs to, or 0 for the current process.
- **coid** The ID of the connection whose flags you want to modify.
- **mask** A bitmap that indicates which bits are to be modified in the flags.
- **bits** The new value of the flags. The flags currently defined include:
  - _NTO_COF_CLOEXEC — close the connection if the process calls an exec*() function to start a new process.

Library:
```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `ConnectFlags()` and `ConnectFlags_r()` kernel calls modify flags associated with the specified connection. These kernel calls don’t block.
These functions are identical except in the way they indicate errors. See the Returns section for details.

You need to initialize the bits that correspond to the flag in both the mask and bits arguments:

- If the bit in the mask is 1, and the bit in the bits is 1, the function turns the flag on.
- If the bit in the mask is 1, and the bit in the bits is 0, the function turns the flag off.
- If bit in the mask is 0, the function doesn’t change the current value of the flag.

**Returns:**

The only difference between these functions is the way they indicate errors:

*ConnectFlags()*

The previous value of the flags associated with the connection.
If an error occurs, the function returns -1 and sets errno.

*ConnectFlags_r()*

The previous value of the flags associated with the connection.
This function does NOT set errno. If an error occurs, the negative of a value from the Errors section is returned.

**Errors:**

- EBADF The coid isn’t a valid connection ID for the process.
- ESRCH The process ID is invalid.

**Classification:**

QNX Neutrino
ConnectFlags(), ConnectFlags_r()  © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ConnectAttach(), fcntl()
ConnectServerInfo(),
ConnectServerInfo_r()
Get information about a server connection

Synopsis:

```c
#include <sys/neutrino.h>

int ConnectServerInfo( pid_t pid,
                       int coid,
                       struct _server_info* info );
```

```c
int ConnectServerInfo_r( pid_t pid,
                          int coid,
                          struct _server_info* info );
```

Arguments:

- `pid`  The process ID of the owner of the connection.
- `coid` The connection ID of the connection.
- `info` A pointer to a _server_info structure where the function can store information about the connection. For more information, see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ConnectServerInfo()` and `ConnectServerInfo_r()` kernel calls get information about the connection `coid` owned by process `pid`, and store it in the structure pointed to by `info`. If the process doesn’t have a connection `coid`, the call scans for the next higher connection and returns it if present. Otherwise, -1 is returned. If you wish to check for the existence of an exact connection, you must compare the returned connection with the `coid` you requested.

These functions are identical except in the way they indicate errors. See the Returns section for details.
ConnectServerInfo(), ConnectServerInfo_r() © 2007, QNX
Software Systems GmbH & Co. KG.

_server_info structure

The _server_info structure that info points to includes at least the following members:
- `uint32_t nd`: The server’s node ID.
- `pid_t pid`: The server’s process ID.
- `int32_t chid`: The server’s channel ID.
- `int32_t scoid`: The server’s connection ID.

`uint32_t nd`

The `nd` (node descriptor) is a temporary numeric description of a remote node. For more information, see the Qnet Networking chapter of the System Architecture guide.

<table>
<thead>
<tr>
<th>To:</th>
<th>Use this function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare two <code>nd</code> objects</td>
<td><code>ND_NODE_CMP()</code></td>
</tr>
<tr>
<td>Convert a <code>nd</code> to text</td>
<td><code>netmgr_ndtostr()</code></td>
</tr>
<tr>
<td>Convert text to a <code>nd</code></td>
<td><code>netmgr_strtond()</code></td>
</tr>
</tbody>
</table>

Returns:

The only difference between these functions is the way they indicate errors:

ConnectServerInfo()

A matched `coid`. If an error occurs, the function returns -1 and sets `errno`.

ConnectServerInfo_r()

A matched `coid`. This function does NOT set `errno`. If an error occurs, the function returns the negative of a value from the Errors section.
Errors:

- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided.
- **EINVAL**: Process \( \text{pid} \) doesn’t have a connection \( \geq \text{coid} \).
- **ESRCH**: The process indicated by \( \text{pid} \) doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `ConnectAttach()`, `ConnectClientInfo()`, `MsgInfo()`, `MsgReceivev()`, `ND_NODE_CMP()`, `netmgr_ndlostr()`, `netmgr_remote_nd()`, `netmgr_strtond()`
**Synopsis:**

```c
#include <math.h>

double copysign ( double x,  
              double y);

float copysignf ( float x,    
              float y );
```

**Arguments:**

- `x`: The number to use the magnitude of.
- `y`: The number to use the sign of.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `copysign()` and `copysignf()` functions return the magnitude of `x` and the sign bit of `y`.

If `x` is `NAN`, the function produces `NAN` with the sign of `y`.

**Returns:**

The magnitude of `x` and the sign bit of `y`.

**Examples:**

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c;
```
copysign(), copysignf()

a = 27.0;
b = -5;
c = copysign(a, b);
printf("The magnitude of %f and sign of %f gives %f\n", 
a, b, c);

return(0);
}

produces the output:

The magnitude of 27.000000 and sign of -5.000000 gives -27.000000

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

significand()
**cos(), cosf()**

Compute the cosine of an angle

**Synopsis:**
```
#include <math.h>

double cos( double x );

float cosf( float x );
```

**Arguments:**

- `x`  
  The angle, in radians, for which you want to compute the cosine.

**Library:**

-libm  
Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the cosine of `x` (specified in radians).

---

An argument with a large magnitude may yield results with little or no significance.

---

**Returns:**

The cosine of `x`.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    double value;

    value = cos( M_PI );
    printf( "value = %f\n", value );

    return EXIT_SUCCESS;
}
```

produces the output:

```
value = -1.000000
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`acos()`, `errno`, `sin()`, `tan()`
**cosh(), coshf()**

*Compute the hyperbolic cosine*

**Synopsis:**

```c
#include <math.h>

double cosh( double x );

float coshf( float x );
```

**Arguments:**

- `x`  
  The angle, in radians, for which you want to compute the hyperbolic cosine.

**Library:**

```c
libm
```

Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the hyperbolic cosine (specified in radians) of `x`. A range error occurs if the magnitude of `x` is too large.

**Returns:**

The hyperbolic cosine of `x`.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", cosh(.5) );
}
```
```c
return EXIT_SUCCESS;
}
```

produces the output:

```
1.127626
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno, sinh(), tanh()`
**Synopsis:**

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat( const char* path,
            mode_t mode );

int creat64( const char* path,
             mode_t mode );
```

**Arguments:**

- `path` The path of the file you want to open.
- `mode` The access permissions that you want to use. For more information, see “Access permissions” in the documentation for `stat()`.

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `creat()` and `creat64()` functions create and open the file specified by `path` with the given `mode`.

Calling `creat()` is the same as:

```c
open( path, O_WRONLY | O_CREAT | O_TRUNC, mode );
```

Similarly, calling `creat64()` is the same as:

```c
open64( path, O_WRONLY | O_CREAT | O_TRUNC | O_LARGEFILE, mode );
```

If `path` exists and is writable, it’s truncated to contain no data, and the existing `mode` setting isn’t changed.

If `path` doesn’t exist, it’s created with the access permissions specified by the `mode` argument. The access permissions for the file or directory are specified as a combination of the bits defined in `<sys/stat.h>`.
Returns:

A file descriptor on success, or -1 if an error occurs (errno is set).

Errors:

EACCES Indicates one of the following permission problems:

- Search permission is denied for one of the components in the path.
- The file specified by path exists, and the permissions specified by mode are denied.
- The file specified by path doesn’t exist, and the file couldn’t be created because write permission is denied for the parent directory.

EBADFSYS While attempting to open path, the file itself or a component of its path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to or while the directory was being updated. The filesystem must be repaired before proceeding.

EBUSY The file specified by path is a block special device that’s already open for writing, or path names a file on a filesystem mounted on a block special device that is already open for writing.

EINTR The call to creat() was interrupted by a signal.

EISDIR The file specified by path is a directory and the file creation flags specify write-only or read/write access.

ELOOP Too many levels of symbolic links.

EMFILE This process is using too many file descriptors.

ENAMETOOLONG

The length of path exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
ENFILE   Too many files are currently open in the system.
ENOENT   Either the path prefix doesn’t exist, or the path argument points to an empty string.
ENOSPC   The directory or filesystem that would contain the new file doesn’t have enough space available to create a new file.
ENOSYS   The creat() function isn’t implemented for the filesystem specified by path.
ENOTDIR  A component of the path prefix isn’t a directory.
EROFS    The file specified by path resides on a read-only filesystem.

Examples:

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main( void )
{
  int filedes;

  filedes = creat( "file",
                   S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
  if( filedes != -1 ) {
    /* process file */

    close( filedes );
    return EXIT_SUCCESS;
  }

  return EXIT_FAILURE;
}
```
**Classification:**

`creat()` is POSIX 1003.1; `creat64()` is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`chsize()`, `close()`, `dup()`, `dup2()`, `eof()`, `errno`, `execl()`, `execlp()`, `execle()`, `execve()`, `execvp()`, `fcntl()`, `fileno()`, `fstat()`, `isatty()`, `lseek()`, `open()`, `read()`, `sopen()`, `stat()`, `tell()`, `write()`, `umask()`
Crypt( )
Encrypt a password

Synopsis:

```
#include <unistd.h>

char * crypt( const char * key,
              const char * salt );
```

Arguments:

- `key` A NUL-terminated string (normally a password typed by a user).
- `salt` A two-character string chosen from the set [a-zA-Z0-9./].
  This function doesn’t validate the values for `salt`, and values outside this range may cause undefined behavior. This string is used to perturb the algorithm in one of 4096 different ways.

Library:

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

  This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `crypt()` function performs password encryption. It’s based on the Data Encryption Standard algorithm, and also includes code to deter key search attempts.

This function checks only the first eight characters of `key`.

You can obtain a 56-bit key by taking the lowest 7 bits of `key`. The 56-bit key is used to repeatedly encrypt a constant string (usually all zeroes).

406 QNX Neutrino Functions and Macros September 10, 2007
crypt()

Returns:
A pointer to the 13-character encrypted value, or NULL on failure. The first two characters of the encrypted value are the salt itself.

Classification:
POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:
The return value points to static data that’s overwritten by each call to crypt().

See also:
encrypt(), getpass(), qnx_crypt(), setkey()
login in the Utilities Reference
Copyright © MINIX Operating System
**ctermid()**

Generate the path name of the current controlling terminal

**Synopsis:**

```c
#include <stdio.h>

char * ctermid( char * s );
```

**Arguments:**

- `s` NULL, or a pointer to a buffer in which the function can store the path name of the controlling terminal. This string should be at least `_ctermid` characters long (see `<stdio.h>`).

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ctermid()` function generates a string that contains the path name of the current controlling terminal for the calling process.

If the argument `s` is NULL, the string is built in a static buffer, and the function returns a pointer to the buffer.

**Returns:**

A pointer to the path name of the controlling terminal, or a pointer to a null string if the function can’t locate the controlling terminal.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "Controlling terminal is %s\n", ctermid( NULL ) );
    return EXIT_SUCCESS;
}
```
**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

The `ctermid()` function isn’t thread-safe if the `s` argument is NULL.

**See also:**

`setsid()`, `ttyname()`
**ctime(), ctime_r()**  
Convert calendar time to local time

**Synopsis:**

```c
#include <time.h>

char* ctime( const time_t* timer );

char* ctime_r( const time_t* timer,
               char* buf );
```

**Arguments:**

- **timer** A pointer to a `time_t` object that contains the time that you want to convert to a string.
- **buf** *(ctime_r() only)* A buffer in which `ctime_r()` can store the resulting string. This buffer must be large enough to hold at least 26 characters.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ctime()` and `ctime_r()` functions convert the time pointed to by `timer` to local time and formats it as a string containing exactly 26 characters in the form:

```
Tue May  7 10:40:27 2002
```

410  QNX Neutrino Functions and Macros  September 10, 2007
This function:  Is equivalent to calling:

\texttt{ctime()} \hspace{1em} \texttt{asctime( localtime( timer ) );}
\texttt{ctime_r()} \hspace{1em} \texttt{asctime_r( localtime( timer ), buf )}

The \texttt{ctime()} function places the result string in a static buffer that’s reused each time you call \texttt{ctime()} or \texttt{asctime()}. Calling \texttt{gmtime()} or \texttt{localtime()} could also change the date in this static buffer.

The result string for \texttt{ctime_r()} is contained in the buffer pointed to by \texttt{buf}.

All fields have a constant width. The newline character ‘\n’ and NUL character ‘\0’ occupy the last two positions of the string.

Whenever the \texttt{ctime()} or \texttt{ctime_r()} functions are called, the \texttt{tzset()} function is also called.

The calendar time is usually obtained by using the \texttt{time()} function. That time is Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

You typically set the time on the computer with the \texttt{date} command to reflect Coordinated Universal Time (UTC), and then use the \texttt{TZ} environment variable or \_CS\_TIMEZONE configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.

Returns:

A pointer to the string containing the formatted local time, or NULL if an error occurs.

Classification:

\texttt{ctime()} is ANSI, POSIX 1003.1; \texttt{ctime_r()} is POSIX 1003.1 TSF
ctime(), ctime_r()

ctime()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

cctime_r()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The asctime() and ctime() functions place their results in a static buffer that’s reused for each call to asctime() or ctime().

See also:

asctime(), asctime_r(), clock(), difftime(), gmtime(), localtime(), localtime_r(), mktime(), strftime(), time(), tzset()

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
daemon()
Run a process in the background

Synopsis:

```c
#include <stdlib.h>

int daemon( int nochdir,
            int noclose );
```

Arguments:

- `nochdir` If this argument is 0, the current working directory is changed to the root directory (`/`).
- `noclose` If this argument is 0, standard input, standard output, and standard error are redirected to `/dev/null`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `daemon()` function allows programs to detach themselves from the controlling terminal and run in the background as system daemons.

This function calls `fork()` and `setsid()`.

The controlling terminal behaves as in Unix System V, Release 4. An `open()` on a terminal device not already associated with another session causes the device to become the controlling terminal for that process.

Returns:

Zero for success, or -1 if an error occurs (`errno` is set).
**daemon()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Classification:**

 Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**Caveats:**

Currently, `daemon()` is supported only in single-threaded applications. If you create a thread and then call `daemon()`, the function returns -1 and sets `errno` to ENOSYS.

**See also:**

`fork()`, `procmgr_daemon()`, `setsid()`
Synopsis:

```c
#include <time.h>

unsigned int daylight;
```

Description:

This global variable has a value of 1 when daylight saving time is supported in this locale, and 0 otherwise. Whenever you call a time function, `tzset()` is called to set the variable, based on the current time zone.

Classification:

POSIX 1003.1 XSI

See also:

- `timezone`, `tzname`, `tzset()`
- “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
**DebugBreak()**

Enter the process debugger

**Synopsis:**

```c
#include <sys/neutrino.h>

void DebugBreak( void );
```

**Library:**

`libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `DebugBreak()` kernel call activates the process debugger if you’re debugging the calling process. If not, it sends a SIGTRAP signal to the process.

**Blocking states**

None.

**Classification:**

QNX Neutrino

| Safety              |     |
|---------------------|--|--|
| Cancellation point  | No |
| Interrupt handler   | No |
| Signal handler      | Yes|
| Thread              | Yes|
Caveats:

If you call DebugBreak() from an interrupt handler, it’ll activate the kernel debugger (if it’s present in your boot image) or send the process a SIGTRAP signal.

See also:

DebugKDBreak(), DebugKDOoutput()
DebugKDBreak()

Enter the kernel debugger

Synopsis:

```
#include <sys/neutrino.h>

void DebugKDBreak( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `DebugKDBreak()` kernel call activates the kernel debugger if it’s present in your boot image. If not, nothing happens.

Blocking states

None.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

```
DebugBreak(), DebugKDOoutput()
```

418 QNX Neutrino Functions and Macros September 10, 2007
DebugKDOoutput()

Print text with the kernel debugger

Synopsis:

```c
#include <sys/neutrino.h>

void DebugKDOoutput( const char* str,
                     size_t size );
```

Arguments:

- `str` The string that you want to print.
- `size` The number of characters to print.

Library:

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `DebugKDBreak()` kernel call causes the kernel debugger to print `size` characters from `str` if the kernel debugger is present in your boot image. If it isn’t in your boot image, nothing happens.

When, where, and how the kernel debugger displays this message depends on which host debugger you’re using.

Blocking states

None.

Classification:

QNX Neutrino

Safety

- Cancellation point: No

continued…
DebugKDOoutput()

Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

DebugBreak(), DebugKDBreak()
delay()

Suspended a calling thread for a given length of time

Synopsis:

```c
#include <unistd.h>

unsigned int delay( unsigned int duration );
```

Arguments:

- `duration`: The number of milliseconds for which to suspend the calling thread from execution.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `delay()` function suspends the calling thread for `duration` milliseconds.

The suspension time may be greater than the requested amount, due to the scheduling of other, higher-priority threads by the system.

Returns:

- 0 for success, or the number of unslept milliseconds if interrupted by a signal.

Errors:

If an error occurs, `errno` is set to:

- `EAGAIN`: No timer resources were available to satisfy the request.
delay()

Examples:

```c
#include <unistd.h>
#include <stdlib.h>

void play_sound( void )
{
    ...
}

void stop_sound( void )
{
    ...
}

int main( void )
{
    play_sound();
    delay( 500 ); /* delay for 1/2 second */
    stop_sound();

    return EXIT_SUCCESS;
}
```

Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

alarm(), errno, nanosleep(), nap(), napms(), sleep()
Synopsis:
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>

int devctl( int filedes,
            int dcmd,
            void * dev_data_ptr,
            size_t n_bytes,
            int * dev_info_ptr );

Arguments:

filedes A file descriptor that you obtained by opening the device.
dcmd A device-specific command for the process managing the open device. The set of valid device-control commands, the associated data interpretation, the returned dev_info_ptr values, and the effect of the command on the device all depend on the device driver.

For specific commands, see the <sys/dcmd_*.h> header files; for general information, see “Device-control commands,” below.
dev_data_ptr Depending on the command, this argument is one of:

- a pointer to a buffer containing data to be passed to the driver
- a receiving area for data coming from the driver
- both of the above
- NULL.
n_bytes The size of the data to be sent to the driver, or the maximum size of the data to be received from the driver. MsgSend() is used to transfer the data.
The `devctl()` function sends the device-specific command `dcmd` to the process managing the device opened as `filedes`. For example, you can send commands to specify properties for devices such as keyboards, sound cards or serial ports.

**Device-control commands**

Use these macros to set up the device-control commands:

- `__DIOF(class, cmd, data)`
  Get information from the device.
- `__DION(class, cmd)`
  A command with no associated data.
- `__DIOT(class, cmd, data)`
  Pass information to the device.
- `__DIOTF(class, cmd, data)`
  Pass some information to the device, and get some from it.

The arguments to these macros are:

- `class` The major category for the command. The device-control commands are divided into the following classes to make organization easier:
- _DCMD_ALL — Common (all I/O servers).
- _DCMD_CAM — Low-level (Common Access Method) devices, such as disks or CD-ROMs.
- _DCMD_CHR — Character devices.
- _DCMD_FSYS, _DCMD_BLK — Filesystem/block I/O managers.
- _DCMD_INPUT — Input devices.
- _DCMD_IP — Internet Protocol.
- _DCMD_MEM — Memory card.
- _DCMD_MISC — Miscellaneous commands.
- _DCMD_MIXER — Mixer (Audio).
- _DCMD_NET — Network devices.
- _DCMD_PHOTON — Photon.
- _DCMD_PROC — Process manager.

`cmd` The specific command in the class.

`data` The type of data to pass to and/or from the device. The `dev_data_ptr` argument to `devctl()` must be a pointer to this type of data, and `n_bytes` is usually the size of this type of data.

The size of the structure that’s passed as the last field to the `__DIO*` macros must be less than $2^{14} = 16K$. Anything larger than this interferes with the upper two directional bits.

Resource managers can use the following macros, which are defined in `<devctl.h>`, when handling commands:

`get_device_command(cmd)`

Extract the class and the specific device command from `cmd` (i.e. strip off the data type and the direction).

`get_device_direction(cmd)`

Get the direction of the command (DEVDIR_TO, DEVDIR_FROM, DEVDIR_TOFROM, or DEVDIR_NONE).
Returns:

- EOK: Success.
- EAGAIN: The `devctl()` command couldn’t be completed because the device driver was in use by another process, or the driver was unable to carry out the request due to an outstanding command in progress.
- EBADF: Invalid open file descriptor, `filedes`.
- EINTR: The `devctl()` function was interrupted by a signal.
- EINVAL: The device driver detected an error in `dev_data_ptr` or `n_bytes`.
- EIO: The `devctl()` function couldn’t complete because of a hardware error.
- ENOSYS: The device doesn’t support the `dcmd` command.
- ENOTTY: The `dcmd` argument isn’t a valid command for this device.
- EPERM: The process doesn’t have sufficient permission to carry out the requested command.

Examples:

Example 1: Setting RTS on a serial port

Here’s a quick example of setting and unsetting RTS (Request to Send) on a serial port:

```c
/* For "devctl()" */
#include <devctl.h>
#include <sys/dcmd_chr.h>

/* For "open()" */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* For Errors */
#include <stdlib.h>
#include <stdio.h>
```
int check_RTS(int fd);

int main(void)
{
    int data = 0, fd, error;
    if((fd = open("/dev/ser2", O_RDONLY)) == -1)
    {
        fprintf(stderr, "Error with open() on /dev/ser2. Make sure exists.\n");
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    check_RTS(fd);
    /* Let's turn ON RTS now. */
    data = _CTL_RTS_CHG | _CTL_RTS;
    if (error = devctl(fd, DCMD_CHR_SERCTL, &data, sizeof(data), NULL))
    {
        fprintf(stderr, "Error setting RTS: %s\n",
                strerror(error));
        exit(EXIT_FAILURE);
    }
    /* RTS should now be ON. */
    check_RTS(fd);
    sleep(2);
    /* Now let's turn RTS OFF. */
    data = _CTL_RTS_CHG | 0;
    if (error = devctl(fd, DCMD_CHR_SERCTL, &data, sizeof(data), NULL))
    {
        fprintf(stderr, "Error setting RTS: %s\n",
                strerror(error));
        exit(EXIT_FAILURE);
    }
    /* RTS should now be OFF. */
    check_RTS(fd);
    close(fd);
    return (1);
}

int check_RTS(int fd)
{
    int data = 0, error;
    /* Let's see if RTS is set, tell devctl() we're requesting
     line status information and devctl() then assigns data
     the line status information for us. Too easy. */
}
if (error = devctl (fd, DCMD_CHR_LINESTATUS, &data, sizeof(data), NULL))
{
    fprintf(stderr, "Error setting RTS: %s\n", strerror ( error ));
    exit(EXIT_FAILURE);
}
if (data & _LINESTATUS_SER_RTS)
    printf("RTS is SET!\n");
else
    printf("RTS is NOT set\n");
return(1);
}

The two main areas of interest are the setting of data and the devctl() call. The data variable is used for both sending and receiving data. When setting RTS, data is assigned a value that’s sent to the device via devctl().

<table>
<thead>
<tr>
<th>If data equals:</th>
<th>RTS is turned:</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CTL_RTS_CHG</td>
<td>_CTL_RTS</td>
</tr>
<tr>
<td>_CTL_RTS_CHG</td>
<td></td>
</tr>
</tbody>
</table>

When checking to see if RTS is set, we call devctl() with dcmd set to the DCMD_CHR_LINESTATUS macro and data containing any value (zero is clean). The devctl() function returns with data containing the Line Status value. This then can be used to determine what lines are set on that device. In our example, we check against _LINESTATUS_SER_RTS.

To find out what values to use with different DCMD_* commands, look in the appropriate <sys/dcmd_* .h> header file. For example, you’ll find macros for the following values under DCMD_CHR_LINESTATUS in <sys/dcmd_chr.h>:

- Serial Port (DTR, RTS, CTS, DSR, RI, CD)
- Keyboard (Scroll/Caps/Num Lock, Shift, CTRL, ALT)
• Parallel Port (No Error, Selected, Paper Out, No Tack, Not Busy)
  The value that’s in the header is a “bitwise &” with the value in data
  to see if the value is high for that line.

Example 2: Cycling through Caps Lock, Num Lock, and Scroll Lock

In the following example, we open the device /dev/kbd and we start
applying changes to the Caps Lock, Scroll Lock, and Num Lock
properties.

The key lines in this example are the same as in the last example; they
focus around the data variable. This value is just a simple integer
value that’s passed into the devctl( ) function. The data variable is
assigned its values by simply performing a bitwise OR to the
predefined values in the </usr/include/sys/dcmd_chr.h> header. Note the values used in the bitwise OR:

• _CONCTL_NUM_CHG (Console Control Num Lock Change)
  ORed together with _CONCTL_NUM (Console Control Num
  Lock) turns on Num Lock.

• _CONCTL_NUM_CHG on its own turns off Num Lock.

<table>
<thead>
<tr>
<th>If data equals:</th>
<th>Num Lock is turned:</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CONCTL_NUM_CHG</td>
<td>_CONCTL_NUM</td>
</tr>
<tr>
<td>_CONCTL_NUM_CHG</td>
<td></td>
</tr>
</tbody>
</table>

This also applies for the other either/or values in the <dcmd_chr.h> header.

/* For "devctl()" */
#include <devctl.h>
#include <sys/dcmd_chr.h>

/* For "open()" */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* For Errors */
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    int data, fd, toggle = 1, error;
    /* Open the device we wish to manipulate. */
    if((fd = open("/dev/kbd", O_RDONLY)) == -1)
        {fprintf(stderr, "Error with open() on /dev/kbd. Make sure exists.\n");
            perror(NULL);
            exit(EXIT_FAILURE);
        }
    while(1)
    {
        switch(toggle)
        {
            case 1:
            {
                /*
                Turn on Num Lock and make sure that
                Caps and Scroll lock are turned off.
                */
                data = (_CONCTL_NUM_CHG | _CONCTL_NUM) | _CONCTL_CAPS_CHG |
                        _CONCTL_SCROLL_CHG;
                break;
            }
            case 2:
            {
                /*
                Turn off Num lock and now turn on Caps Lock
                (Scroll lock is already off).
                */
                data = _CONCTL_NUM_CHG | (_CONCTL_CAPS_CHG | _CONCTL_CAPS);
                break;
            }
            case 3:
            {
                /*
                Turn off Caps lock and turn on Scroll lock
                (Num lock is already off).
                */
                data = _CONCTL_CAPS_CHG | (_CONCTL_SCROLL_CHG | _CONCTL_SCROLL);
                toggle = 0;
                break;
            }
        }
    }
    /* Explanation below. */
    if (error = devctl (fd, DCMD_CHR_SERCTL, &data, sizeof(data), NULL))
        {fprintf(stderr, "Error setting KBD: %s\n",
                    strerror (error));
            exit(EXIT_FAILURE);
        }
    sleep(1);
    toggle++;
}
devctl()

Here's a quick explanation of the above devctl() call:

```
devctl (fd, DCMD_CHR_SERCTL, &data, sizeof(data), NULL)
```

The first parameter, `fd`, is the file descriptor of the device that’s being changed. The second parameter is the device class that’s being changed. In this case, it’s a character device `DCMD_CHR`, with a “subclass” of `SERCTL`. The third parameter is the data variable; this is the ORed value.

**Example 3: Duration example**

In this code, `tcdropline()`, which is used to disconnect a communications line, uses `devctl()` (this is the actual source code, `tcdropline()` is a standard library function):

```
#include <termios.h>
#include <devctl.h>
#include <errno.h>
#include <sys/dcmd_chr.h>

int tcdropline(int fd, int duration) {
    int error;
    duration = ((duration ? duration : 300) << 16) | _SERCTL_DTR_CONT | 0;
    if(error = devctl(fd, DCMD_CHR_SERCTL, &duration, sizeof duration, 0) == -1) {
        if(error == ENOSYS) {
            errno = ENOTTY;
        } else {
            return -1;
        }
    }
    return 0;
}
```

**Classification:**

QNX Neutrino


Caveats:

When `devctl()` fails, the effect of the failed command depends on the device driver. The corresponding data might be transferred, partially transferred, or not transferred at all.

The `devctl()` function was originally part of the POSIX 1003.1d draft standard; but it was deprecated in the IEEE Approved Draft 10 standard.

See also:

`close()`, `open()`, `read()`, `write()`
**Synopsis:**

```c
#include <time.h>

double difftime( time_t time1,  
                time_t time0 );
```

**Arguments:**

- `time1, time0`  
The times to compare, expressed as `time_t` objects.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `difftime()` function calculates the difference between the calendar times specified by `time1` and `time0`:

`time1 - time0`

**Returns:**

The difference between the two times (in seconds).

**Examples:**

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void compute( void )
{
    int i, j;
    for( i = 1; i <= 20; i++ ) {
        for( j = 1; j <= 20; j++ ) {
            printf( "%3d " , i * j );
        }
        printf( "\n" );
    }
}
```
int main( void )
{
    time_t start_time, end_time;
    start_time = time( NULL );
    compute();
    end_time = time( NULL );
    printf( "Elapsed time: %f seconds\n",
            difftime( end_time, start_time ) );
    return EXIT_SUCCESS;
}

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asctime(), clock(), ctime(), gmtime(), localtime(), mktime(), strftime(), time(), tzset()
Synopsis:

#include <dirent.h>

int dircntl( DIR * dir,
            int cmd,
            ... );

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>Provide control for this directory.</td>
</tr>
</tbody>
</table>
| cmd | At least the following values are defined in <dirent.h>:

- D_GETFLAG — retrieve the flags associated with the directory referenced by dir. For more information, see “Flag values,” below.
- D_SETFLAG — set the flags associated with the directory referenced by dir to the value given as an additional argument. The new value can be any combination of the flags described in “Flag values,” below.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The dircntl() function provides control over the open directory referenced by the dir argument. This function behaves in a manner similar to the file control function, fcntl().

Flag values

D_FLAG_FILTER

Filter out duplicate name entries that may occur due to the union filesystem during a readdir() operation.
**dircntl()**

© 2007, QNX Software Systems GmbH & Co. KG.

D_FLAG_STAT

Indicate to servers that they should attempt to return extra stat() information as part of the readdir() operation.

**Returns:**

The return value depends on the value of cmd:

**D_GETFLAG**  The flags associated with the directory, or -1 if an error occurs (errno is set).

**D_SETFLAG**  0 for success, or -1 if an error occurs (errno is set).

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main(int argc, char **argv) {
    DIR *dp;
    int ret;

    if(!(dp = opendir("/"))) {
        exit(EXIT_FAILURE);
    }

    /* Display the flags that are set on the directory by default*/
    if((ret = dircntl(dp, D_GETFLAG)) == -1) {
        exit(EXIT_FAILURE);
    }

    if(ret & D_FLAG_FILTER) {
        printf("Directory names are filtered
");
    } else {
        printf("Directory names are not filtered
");
    }

    if(ret & D_FLAG_STAT) {
        printf("Servers asked for extra stat information
");
    } else {
        printf("Servers not asked for extra stat information
");
    }

closedir(dp);
```

return 0;
}

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`fcntl()`, `opendir()`
dirname()

Find the parent directory part of a file pathname

Synopsis:

#include <libgen.h>

char *dirname( char *path );

Arguments:

path The string to parse.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The dirname() function takes a pointer to a character string that contains a pathname, and returns a pointer to a string that’s a pathname of the parent directory of that file. Trailing “/” characters in the path aren’t counted as part of the path.

If the path doesn’t contain a “/” character, or path is a NULL pointer or points to an empty string, then dirname() function returns a pointer to the string "." (dot).

Together the dirname() and basename() functions yield a complete pathname. The expression dirname(path) obtains the pathname of the directory where basename(path) is found.

The dirname() function might modify the string pointed to by path, and can return a pointer to static storage.

Returns:

A pointer to a string that’s the parent directory of path. If path is a NULL pointer or points to an empty string, a pointer to a string "." is returned.
_examples:

<table>
<thead>
<tr>
<th>String input</th>
<th>String output</th>
</tr>
</thead>
<tbody>
<tr>
<td>“/usr/lib”</td>
<td>“/usr”</td>
</tr>
<tr>
<td>“/usr/”</td>
<td>“usr”</td>
</tr>
<tr>
<td>“/”</td>
<td>“/”</td>
</tr>
<tr>
<td>“.”</td>
<td>“.”</td>
</tr>
<tr>
<td>“..”</td>
<td>“..”</td>
</tr>
</tbody>
</table>

The following code fragment reads a pathname, changes the current working directory to the parent directory, and opens the file:

```c
char path[BUFF_SIZE], *pathcopy;
int fd;

fgets(path, BUFF_SIZE, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

_classification:

POSIX 1003.1 XSI

_Safety_

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

basename()
**dispatch_block()**

Block while waiting for an event

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_context_t * dispatch_block
    ( dispatch_context_t * ctp );
```

**Arguments:**

- `ctp` A pointer to a `dispatch_context_t` structure that defines the dispatch context.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `dispatch_block()` function blocks while waiting for an event (e.g. message or signal) that’s registered using one of the attach functions, `message_attach()`, `pulse_attach()`, `resmgr_attach()`, or `select_attach()`. (The `sigwait_attach()` function isn’t currently implemented.)

<table>
<thead>
<tr>
<th>If the type of blocking is:</th>
<th><code>dispatch_block()</code> does a:</th>
</tr>
</thead>
<tbody>
<tr>
<td>message (resmgr, message, select)</td>
<td><code>MsgReceive()</code></td>
</tr>
<tr>
<td>signal</td>
<td><code>SignalWaitinfo()</code></td>
</tr>
</tbody>
</table>

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.
dispatch_block()

Returns:

A dispatch context that’s passed in by dispatch_context_alloc(). or
NULL if an error occurs (errno is set).

Errors can occur when the blocking kernel call returns with an error,
for example, due to the delivery of a signal.

In the case of a timeout, a valid ctp is returned, but either the
ctp->message_context.rcvid or ctp->sigwait_context.signo is set to
-1.

If a non-NULL context pointer is returned, it could be different from
the one passed in, as it’s possible for the ctp to be reallocated to
a larger size. In this case, the old ctp is no longer valid. However, if
NULL is returned (for example, because a signal interrupted the
MsgReceive()), the old context pointer is still valid. Typically, a
resource manager would target signals to a thread dedicated to
handling signals. However, if a signal can be targeted to the thread
doing dispatch_block(), you could use the following code in this
situation:

dispatch_context_t *ctp, *new_crp;
ctp = dispatch_context_alloc( ... );
while (1) {
    new_crp = dispatch_block( ctp );
    if ( new_crp ) {
        ctp = new_crp
    }
    else {
        /* handle the error condition */
        :
    }
}

Errors:

EFAULT A fault occurred when the kernel tried to access the
buffers.

EINVAL The call was interrupted by a signal.
**dispatch_block()**

- **EINVAL**: Invalid arguments passed to dispatch_block().
- **ENOMEM**: Insufficient memory to allocate internal data structures.

See also the error constants returned in *MsgReceive()* and *SignalWaitinfo()*.

**Examples:**

```c
#include <sys/dispatch.h>

int main( int argc, char **argv ) {
    dispatch_context_t *ctp;

    ...

    for(;;) {
        if( ctp = dispatch_block( ctp ) ) {
            dispatch_handler( ctp );
        }
    }
}
```

For examples using the dispatch interface, see *dispatch_create()*,
*message_attach()*,*resmgr_attach()*,*thread_pool_create()*.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

dispatch_context_alloc(), dispatch_handler(), dispatch_timeout(),
dispatch_unblock()

“Components of a resource manager” in the Writing a Resource
Manager chapter of the Neutrino Programmer’s Guide
Return a dispatch context

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_context_t * dispatch_context_alloc
    ( dispatch_t * dpp );
```

Arguments:

- **dpp**: A dispatch handle created by `dispatch_create()`.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dispatch_context_alloc()` function returns a dispatch context pointer. The function is passed in the handle `dpp` from `dispatch_create()`. The dispatch context is used by dispatch to do its work. It’s passed as an argument to `dispatch_block()` and `dispatch_handler()`.

The `dispatch_context_alloc()` function fails if you haven’t attached any events to dispatch yet (e.g. you didn’t call `message_attach()`, `resmgr_attach()`, or `select_attach()`). The dispatch library can’t allocate a proper context until it knows what kind of events you want to block.

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.
**dispatch_context_alloc()** © 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

A pointer to a dispatch context, or NULL if an error occurs (*errno* is set).

**Errors:**

- **EINVAL** No events were attached.
- **ENOMEM** Insufficient memory to allocate context.

**Examples:**

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    dispatch_context_t *ctp;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
    avellip;
    ctp = dispatch_context_alloc( dpp );
    avellip;

    return EXIT_SUCCESS;
}
```

For examples using the dispatch interface, see *dispatch_create()*, *message_attach()**, resmgr_attach()**, and *thread_pool_create()*.

**Classification:**

QNX Neutrino
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `dispatch_block()`, `dispatch_context_free()`, `dispatch_create()`, `dispatch_handler()`, `dispatch_unblock()`
- “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino *Programmer’s Guide*
Free a dispatch context

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void dispatch_context_free(
    dispatch_context_t * ctp );
```

Arguments:

- `ctp`: A pointer to a `dispatch_context_t` structure that was allocated by `dispatch_context_alloc()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dispatch_context_free()` function frees the given dispatch context.

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.

Examples:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    dispatch_context_t *ctp;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "Unable to allocate dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }
    dispatch_context_alloc
    ...}
```
dispatch_context_free()

:

ctp = dispatch_context_alloc( dpp );

:

dispatch_context_free ( ctp );
return EXIT_SUCCESS;
}

See dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create() for examples using the dispatch interface.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

dispatch_context_alloc()

“Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide
**dispatch_create()**

Allocate a dispatch handle

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

dispatch_t *dispatch_create( void );
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `dispatch_create()` function allocates and initializes a dispatch handle. The attach functions are:

- `message_attach()`
- `pulse_attach()`
- `resmgr_attach()`
- `select_attach()`

If you wish, you can do a `resmgr_attach()` with a NULL path. This has the effect of initializing dispatch to receive messages and creates the channel among other things.

A channel is created only when you first attach something that requires a channel (indicating you will block receiving messages).

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino *Programmer’s Guide*. 

---

QNX Neutrino Functions and Macros  September 10, 2007
Returns:

A handle to a dispatch structure, or NULL if an error occurs.

The dispatch structure, `dispatch_t`, is an opaque data type; you can’t access its contents directly.

Errors:

- ENOMEM: Insufficient memory to allocate context.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <fcntl.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int my_func( select_context_t *ctp, int fd, unsigned flags, void *handle ) {
    int i, c;

    /* Do some useful stuff with data */
    i = read( fd, &c, 1 );
    fprintf( stderr, "activity on fd %d: read char %c,
                return code %d\n", fd, c, i );
}

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    dispatch_context_t *ctp;
    select_attr_t attr;
    int fd, fd2;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \n
dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }

    if( argc ≤ 2 || (fd = open( argv[1],
        O_RDWR | O_NONBLOCK )) == -1 ) {
        return EXIT_FAILURE;
    }
```
if( argc ≤ 2 | | (fd2 = open( argv[2],
0_RDWR | O_NONBLOCK )) == -1 ) {
    return EXIT_FAILURE;
}
select_attach( dpp, &attr, fd,
SELECT_FLAG_READ | SELECT_FLAG_REARM, my_func, NULL );
select_attach( dpp, &attr, fd2,
SELECT_FLAG_READ | SELECT_FLAG_REARM, my_func, NULL );
ctp = dispatch_context_alloc( dpp );

for(;;) {
    if( ctp = dispatch_block( ctp ) ) {
        dispatch_handler( ctp );
    }
} return EXIT_SUCCESS;

For more examples using the dispatch interface, see
message_attach(), resmgr_attach(), and thread_pool_create().

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

dispatch_block(), dispatch_context_alloc(), dispatch_destroy(),
dispatch_handler(), dispatch_timeout(), dispatch_unblock()  
message_attach(), pulse_attach(), resmgr_attach(), select_attach()

“Components of a resource manager” in the Writing a Resource
Manager chapter of the Neutrino Programmer’s Guide
**dispatch_destroy()**

*Destroy a dispatch handle*

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int dispatch_destroy( dispatch_t *dpp );
```

**Arguments:**

- `dpp` A dispatch handle created by `dispatch_create()`.

**Library:**

libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The function `dispatch_destroy()` destroys the given dispatch handle.

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.

**Returns:**

- `0` Success.
- `-1` An error occurred (`errno` is set).

**Errors:**

- `EINVAL` The dispatch handle, `dpp`, is invalid.
dis\textunderscore destroy() © 2007, QNX Software Systems GmbH & Co. KG.

Examples:

\begin{verbatim}
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int destroyed;

    if( ( dpp = dispatch_create() ) == NULL ) {  
        fprintf( stderr, "Unable to allocate dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    ...

    if ( (destroyed = dispatch_destroy ( dpp )) == -1 ) {  
        fprintf ( stderr, "Dispatch wasn’t destroyed, bad dispatch handle %d.\n", dpp);
        return EXIT_FAILURE;
    }
    /* else dispatch was destroyed */

    ...
    return EXIT_SUCCESS;
}
\end{verbatim}

For examples using the dispatch interface, see \texttt{dispatch\_create()}, \texttt{message\_attach()}, \texttt{resmgr\_attach()}, and \texttt{thread\_pool\_create()}.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

\textit{dispatch\_create()}

“Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino \textit{Programmer’s Guide}
**dispatch_handler()**

Handle events received by dispatch_block()

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int dispatch_handler( dispatch_context_t *ctp );
```

**Arguments:**

- `ctp` A pointer to a `dispatch_context_t` structure that was allocated by `dispatch_context_alloc()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `dispatch_handler()` function handles events received by `dispatch_block()`. This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.

Depending on the blocking type, `dispatch_handler()` does one of the following:

- Calls the `message_`* subsystem. A search is made (based upon the message type or pulse code) for a matching function (that was attached with `message_attach()` or `pulse_attach()`). If a match is found, the attached function is called.

- If the message type is in the range handled by the resource manager (e.g. I/O messages) and pathnames were attached using `resmgr_attach()`, then the `resmgr_`* subsystem is called and handles the resource manager messages.

- If a pulse is received, it may be dispatched to the `resmgr_`* subsystem if it’s one of the codes (unblock and disconnect pulses).
handled by the resource manager. If a \texttt{select\_attach()} was done and the pulse matches the one used by \texttt{select\_attach()}, then the \texttt{select\_*} subsystem is called and dispatches that event.

- If a message is received, and no matching handler is found for that message type, \texttt{MsgError()} returns ENOSYS to the sender.

- If a \texttt{SignalWaitinfo()} blocking type is used, then a search is made based upon the signal number for a matching function attached by the program (using the \texttt{sigwait\_attach()} function, not currently implemented). If a match is found, that function is called.

\section*{Returns:}

0 \hspace{1em} Success.

\textbf{-1} \hspace{1em} One of the following occurred:

- The message was a \texttt{\_PULSE\_CODE\_THREADDEATH} pulse message (see \texttt{ChannelCreate()}) for which there’s no default handler function.

- The message length was less than 2 bytes. A 2-byte message type is required at the beginning of the message so that a handler function can be found or identified.

- The message wasn’t in native endian format and there were no handler functions that specified \texttt{MSG\_FLAG\_CROSS\_ENDIAN} on this range, even though a handler was registered for it using \texttt{message\_attach()}. The \texttt{MSG\_FLAG\_CROSS\_ENDIAN} flag wasn’t given to \texttt{message\_attach()}.  

- A handler was found for the message, but the handler determined that there was a problem.

In any case, if the message wasn’t a pulse, then the client will be replied to with an appropriate \texttt{errno}.  

\section*{September 10, 2007 QNX Neutrino Functions and Macros 457}
Examples:

```c
#include <stdlib.h>
#include <sys/dispatch.h>

int main(int argc, char **argv) {
    dispatch_context_t *ctp;

    ...
    for(;;) {
        if( ctp = dispatch_block( ctp ) ) {
            dispatch_handler( ctp );
        }
    }
    return EXIT_SUCCESS;
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Read Caveats</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

This function might or might not be a cancellation point, depending on the implementation of the handler.
See also:

\textit{dispatch\_block()}, \textit{dispatch\_create()}, \textit{dispatch\_timeout()}

“Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino \textit{Programmer’s Guide}
**dispatch_timeout()**

Set a timeout

---

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int dispatch_timeout( dispatch_t *dpp,
                      struct timespec *reltime );
```

**Arguments:**

- `dpp` A dispatch handle created by `dispatch_create()`.
- `reltime` A pointer to a `timespec` structure that specifies the relative time of the timeout.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The function `dispatch_timeout()` sets a timeout that's used when blocking with `dispatch_block()`.

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.

**Returns:**

- 0 Success.
- -1 An error occurred.
**Examples:**

```c
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    dispatch_t *dpp;
    struct timespec time_out;
    int timedout;
    time_out.tv_sec = 1;
    time_out.tv_nsec = 2;

    if( (dpp = dispatch_create()) == NULL ) {
        fprintf(stderr, "%s: Unable to allocate dispatch handle.\n", argv[0]);
        return EXIT_FAILURE;
    }

    ... 

    if ( (timedout = dispatch_timeout(dpp, &time_out)) == -1 ) {
        fprintf(stderr, "Couldn’t set timeout");
        return EXIT_FAILURE;
    } /* else successful timeout set */

    ... 
    return EXIT_SUCCESS;
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: No

*continued...*
### dispatch_timeout()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`dispatch_block()`, `dispatch_create()`, `dispatch_handler()`, `dispatch_unblock()`

timespec

“Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino *Programmer’s Guide*
**dispatch_unblock()**

Unblock all of the threads that are blocked on a dispatch handle

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void dispatch_unblock( dispatch_context_t * ctp );
```

**Arguments:**

- **ctp** A pointer to a `dispatch_context_t` structure that defines the dispatch context.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

This routine tries to unblock all of the threads that are blocked on the given dispatch handle. You should use this function in the thread pool structure as the unblock function pointer so that `thread_pool_control()` will behave properly.

Currently, this function unblocks only channel resources.

This function is part of the dispatch layer of a resource manager. For more information, see “Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.

**Examples:**

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.
**dispatch_unblock()**

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*dispatch_block(), dispatch_context_alloc(), dispatch_handler(), dispatch_timeout()*

“Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino *Programmer's Guide*
Synopsis:

```c
#include <stdlib.h>

div_t div( int numer,
           int denom );
```

Arguments:

- `numer`  The numerator in the division.
- `denom`  The denominator.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `div()` function calculates the quotient and remainder of the division of `numer` by `denom`.

Returns:

A `div_t` structure containing the quotient and remainder:

```c
typedef struct {
    int quot;  /* quotient */
    int rem;  /* remainder */
} div_t;
```

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

void print_time( int seconds )
{
    div_t min_sec;

    min_sec = div( seconds, 60 );
}
```
```c
int main( void ) {
    print_time( 130 );
    return EXIT_SUCCESS;
}
```

produces the output:

It took 2 minutes and 10 seconds

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`ldiv()`
dladdr()

Translate an address to symbolic information

Synopsis:

```c
#include <dlfcn.h>

int dladdr( void *address,
            Dl_info *dlip );
```

Arguments:

- `address` The address for which you want symbolic information.
- `dlip` A pointer to a `Dl_info` structure where the function can store the symbolic information. Your application must allocate the space for this structure; `dladdr()` fills in the members, based on the specified address.

The `Dl_info` structure includes the following members:

- `const char * dli_fname` — a pointer to the filename of the object containing `address`.
- `void *dli_fbase` — the base address of the object containing `address`.
- `const char *dli_sname` — a pointer to the symbol name nearest the specified `address`. This symbol is either at `address`, or is the nearest symbol with a lower address.
- `void *dli_saddr` — the actual address of the `dli_sname` symbol.

If `dladdr()` can’t find a symbol that describes the specified `address`, the function sets `dli_sname` and `dli_saddr` to `NULL`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**Description:**

The `dladdr()` function determines whether the specified address is located within one of the objects that make up the calling application’s address space.

The `dladdr()` function is available only to dynamically linked processes.

**Returns:**

0 if the specified address can’t be matched, or nonzero if it could be matched.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The `D1_info` pointers may become invalid if objects are removed via `dlclose()`.

There’s no way to determine which symbol you’ll get if multiple symbols are mapped to the same address.
See also:

\textit{dlclose()}, \textit{dlerror()}, \textit{dlopen()}, \textit{dlsym()}
dlclose()  © 2007, QNX Software Systems GmbH & Co. KG.

Close a shared object

Synopsis:

```
#include <dlfcn.h>

int dlclose( void *handle );
```

Arguments:

- `handle`: A handle for a shared object, returned by `dlopen()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dlclose()` function disassociates a shared object opened by `dlopen()` from the calling process. An object’s symbols are no longer available after it’s been closed with `dlclose()`. All objects loaded as a result of the closed objects dependencies are also closed.

The `handle` argument is the value returned by a previous call to `dlopen()`.

Errors:

If an error occurs, more detailed diagnostic information is available from `dlerror()`.
**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

An object won’t be removed from the address space until all references to that object (via `dlopen()` or dependencies from other objects) have been closed.

Referencing a symbol in a closed object can cause undefined behavior.

**See also:**

`dladdr()`, `dlerror()`, `dlopen()`, `dlsym()`
dlerror()

Get dynamic loading diagnostic information

Synopsis:

```c
#include <dlfcn.h>

char *dlerror( void );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dlerror()` function returns a NULL-terminated string (with no trailing newline) describing the last error that occurred during a call to one of the `dl*()` functions. If no errors have occurred, `dlerror()` returns NULL.

```
The `dlopen()` function is available only to dynamically linked processes.
```

Returns:

A pointer to an error description, or NULL.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

dlladdr(), dlclose(), dlopen(), dsym()
**dlopen()**

Gain access to an executable object file

**Synopsis:**

```c
#include <dlfcn.h>

void * dlopen( const char * pathname,
               int mode );
```

**Arguments:**

- **pathname**: NULL, or the path to the executable object file that you want to access.
- **mode**: Flags that control how `dlopen()` operates; see “The `mode`,” below.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `dlopen()` function gives you direct access to the dynamic linking facilities by making the executable object file specified in `pathname` available to the calling process. It returns a handle that you can use in subsequent calls to `dlsym()` and `dlclose()`.
The `dlopen()` function is available only to a dynamically-linked process. A statically-linked process (one where `libc` is linked statically) can’t call `dlopen()` because a statically-linked executable:

- doesn’t export any of its symbols
- can’t export the required structure for libraries to link against
- can’t fill structures at startup needed to load subsequent shared objects.

Any dependencies recorded within `pathname` are loaded as part of the `dlopen()` call. These dependencies are searched in load-order to locate any additional dependencies. This process continues until all of the dependencies for `pathname` have been satisfied. This dependency tree is called a group.

If `pathname` is NULL, `dlopen()` provides a handle to the running process’s global symbol object. This provides access to the symbols from the original program image file, the dependencies it loaded at startup, plus any objects opened with `dlopen()` calls using the `RTLDGLOBAL` flag. This set of symbols can change dynamically if the application subsequently calls `dlopen()` using `RTLDGLOBAL`.

You can use `dlopen()` any number of times to open objects whose names resolve to the same absolute or relative path name; the object is loaded into the process’s address space only once.

In order to find the shared objects, the following directories or paths are searched in order:

- **RPATH**
- **LD_LIBRARY_PATH**
- **_CS_LIBPATH**.

Note that `LD_LIBRARY_PATH` is ignored if the binary is `setuid` and the `euid` is not the same as the `uid` of the user running the binary. This is done for security purposes.
The above directories are set as follows:

- The **RPATH** value is set up when binary is linked, using the `ld` command line option `-rpath`. See `ld` for details.

- The **LD_LIBRARY_PATH** is generally set up by other startup script, either in the boot image or by a secondary script. For example, on self hosted QNX system, it is setup by `ph` script. It is not part of any default environment.

- **_CS_LIBPATH** is populated by the kernel, and the default value is based on the **LD_LIBRARY_PATH** value of the `procnto` command line in the boot image. Note that, you may use `getconf` utility to inspect this value and `setconf` to set this value. For example:

  ```bash
  setconf _CS_LIBPATH 'getconf _CS_LIBPATH':/new/path
  ```

When loading shared objects, the application should open a specific version instead of relying on the version pointed to by a symbolic link.

### The mode

The **mode** argument indicates how `dlopen()` operates on `pathname` when handling relocations, and controls the visibility of symbols found in `pathname` and its dependencies.

The **mode** argument is a bitwise-OR of the constants described below. Note that the relocation and visibility constants are mutually exclusive.

### Relocation

When you load an object by calling `dlopen()`, the object may contain references to symbols whose addresses aren’t known until the object has been loaded; these references must be relocated before accessing the symbols. The **mode** controls when relocations take place, and can be one of:
References to data symbols are relocated when the object is loaded. References to functions aren’t relocated until that function is invoked. This improves performance by preventing unnecessary relocations.

All references are relocated when the object is loaded. This may waste cycles if relocations are performed for functions that never get called, but this behavior could be useful for applications that need to know that all symbols referenced during execution are available as soon as the object is loaded.

make the object’s global symbols available only to objects in the same group.

The program’s image and any objects loaded at program startup have a mode of RTLD_GLOBAL; the default mode for objects acquired with dlopen() is RTLD_LOCAL. A local object may be part of the dependencies for more than one group; any object with a RTLD_LOCAL mode referenced as a dependency of an object with a RTLD_GLOBAL mode is promoted to RTLD_GLOBAL.

Visibility

The following mode bits determine the scope of visibility for symbols loaded with dlopen():

RTLD_GLOBAL

Make the object’s global symbols available to any other object. Symbol lookup using dlopen( 0, mode ) and an associated dlsym() are also able to find the object’s symbols.

RTLD_LOCAL

Make the object’s global symbols available only to objects in the same group.

Visibility
Objects loaded with `dlopen()` that require relocations against global symbols can reference the symbols in any RTLDGLOBAL object. You can OR the `mode` with the following values to affect symbol scope:

**RTLD_GROUP**

Only symbols from the associated group are available. All dependencies between group members must be satisfied by the objects in the group.

**RTLD_WORLD**

Only symbols from RTLDGLOBAL objects are available.

The default `mode` is RTLD_WORLD | RTLD_GROUP.

If you specify RTLD_WORLD without RTLD_GROUP, `dlopen()` doesn’t load any of the DLL’s dependencies.

### Symbol Resolution

When resolving the symbols in the shared object, the runtime linker searches for them in the dynamic symbol table using the following order:

By default:

1. main executable
2. the shared object being loaded
3. all other loaded shared objects that were loaded with the RTLDGLOBAL flag.

When `-Bsymbolic` is specified:

1. the shared object being loaded
2. main executable
3. all other loaded shared objects that were loaded with the RTLDGLOBAL flag.
For executables, the dynamic symbol table typically contains only those symbols that are known to be needed by any shared libraries. This is determined by the linker when the executable is linked against a shared library.

Since you don’t link your executable against a shared object that you load with `dlopen()`, the linker can’t determine which executable symbols need to be made available to the shared object.

If your shared object needs to resolve symbols in the executable, then you may force the linker to make all of the symbols in the executable available for dynamic linking by specifying the `-E` linker option. For example:

```bash
qcc -Vgcc_ntox86 -Wl,-E -o main main.o
```

Shared objects always place all their symbols in dynamic symbol tables, so this option isn’t needed when linking a shared object.

**Returns:**

A handle to the object, or NULL if an error occurs.

Don’t interpret the value of this handle in any way. For example, if you open the same object repeatedly, don’t assume that `dlopen()` returns the same handle.

**Errors:**

If an error occurs, more detailed diagnostic information is available from `dlerror()`.

**Environment variables:**

- **DL_DEBUG**
  Display debugging information about the libraries as they’re opened.

- **LD_LIBRARY_PATH**
  The `LD_LIBRARY_PATH` environment variable is searched for any dependencies required by `pathname`. 
**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

Some symbols defined in executables or shared objects might not be available to the runtime linker. The symbol table created by `ld` for use by the runtime linker might contain a subset of the symbols defined in the object.

**See also:**

`dladdr()`, `dlclose()`, `dlerror()`, `dlsym()`

`ld`, `qcc` in the *Utilities Reference*
Synopsis:

```c
#include <dlfcn.h>

void* dlsym( void* handle,
             const char* name );
```

Arguments:

- `handle`: Either a handle for a shared object, returned by `dlopen()`, or the special flag, RTLD_DEFAULT.
- `name`: The name of the symbol that you want to find in the shared object.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dlsym()` function lets a process obtain the address of the symbol specified by `name` defined in a shared object.

The `dlsym()` function is available only to dynamically linked processes.

If `handle` is a handle returned by `dlopen()`, you must not have closed that shared object by calling `dlclose()`. The `dlsym()` functions also searches for the named symbol in the objects loaded as part of the dependencies for that object.

If `handle` is RTLD_DEFAULT, `dlsym()` searches all objects in the current process, in load-order.

In the case of RTLD_DEFAULT, if the objects being searched were loaded with `dlopen()`, `dlsym()` searches the object only if the caller is part of the same dependency hierarchy, or if the object was loaded with global search access (using the RTLD_GLOBAL mode).
**Returns:**

A pointer to the named symbol for success, or NULL if an error occurs.

**Errors:**

If an error occurs, more detailed diagnostic information is available from `dlerror()`. 

**Examples:**

Use `dlsym()` to find a function pointer and a pointer to a global variable in a shared library:

```c
typedef int (*foofunc)( int );

void* handle;
int* some_global_int;
foofunc brain;

/* Open a shared library. */
handle = dlopen( "/usr/nto/x86/lib/libfoo.so.1", RTLD_NOW );

/* Find the address of a function and a global integer. */
brain = (foofunc)dlsym( handle, "takeover_world" );
some_global_int = (int*)dlsym( handle, "my_global_int" );

/* Invoke the function and print the int. */
x = (*brain)( 5 );
printf( "that global is %d\n", *some_global_int );
```

Check to see if a function is defined, and call it if it is:

```c
typedef int (*funcptr)( void );

funcptr funk = NULL;

funk = (funcptr)dlsym( RTLD_DEFAULT, "get_funky" );
if( funk != NULL ) {
    (*funk)();
}
```
Classification:

POSIX 1003.1 XSI

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Function pointers are a pain; use `typedef`s to help preserve your sanity.

See also:

`dladdr()`, `dlclose()`, `dlerror()`, `dlopen()`
**dn_comp()**

Compress an Internet domain name

**Synopsis:**
```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp( const char * exp_dn,
             u_char * comp_dn,
             int length,
             u_char ** dnptrs,
             u_char ** lastdnptr );
```

**Arguments:**
- `exp_dn` The Internet domain name you want to compress.
- `comp_dn` A buffer where the function can store the compressed name.
- `length` The size of the array that `comp_dn` points to.
- `dnptrs` NULL, or an array of pointers to previously compressed names in the current message; see below.
- `lastdnptr` NULL, or the limit of the array specified by `dnptrs`.

**Library:**
```
libsocket
```
Use the `-l socket` option to `qcc` to link against this library.

**Description:**
The `dn_comp()` routine is a low-level routine used by `res_query()` to compress an Internet domain name. This routine compresses the domain name `exp_dn` and stores it in `comp_dn`.

The compression uses an array of pointers, `dnptrs`, to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The limit
to the array is specified by lastdnptr. As a side effect, \texttt{dn\_comp()} updates the list of pointers for labels inserted into the message as the name is compressed. If \texttt{dnptrs} is NULL, names aren’t compressed. If \texttt{lastdnptr} is NULL, the list of labels isn’t updated.

\textbf{Returns:}

The size of the compressed domain name, in bytes, or -1 if an error occurs.

\textbf{Classification:}

Unix

\begin{tabular}{l|c}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}

\textbf{See also:}

\texttt{dn\_expand()}, \texttt{gethostbyname()}, \texttt{res\_init()}, \texttt{res\_mkquery()}, \texttt{res\_query()}, \texttt{res\_search()}, \texttt{res\_send()}

\texttt{/etc/resolv.conf, hostname} in the \textit{Utilities Reference}

\textit{RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035}
**dn_expand()**

Expand a compressed Internet domain name

**Synopsis:**
```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand( const u_char * msg,
               const u_char * eomorig,
               const u_char * comp_dn,
               char * exp_dn,
               int length );
```

**Arguments:**
- `msg` A pointer to the beginning of the message that contains the compressed name.
- `eomorig` A pointer to the first location after the message.
- `comp_dn` The compressed name that you want to expand.
- `exp_dn` A buffer where the function can store the expanded name.
- `length` The length of the array specified by `exp_dn`.

**Library:**
- `libsocket`
  
  Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `dn_expand()` function is a low-level routine used by `res_query()` to expand the compressed domain name, `comp_dn`, to a full domain name.

The compressed name is contained in a query or reply message.
Returns:

The size of the compressed domain name (not the expanded name), in bytes, or -1 if an error occurs.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `dn_comp()`, `gethostbyname()`, `res_init()`, `res_mkquery()`, `res_query()`, `res_search()`, `res_send()`
- `/etc/resolv.conf`, `hostname` in the Utilities Reference
- `RFC 974`, `RFC 1032`, `RFC 1033`, `RFC 1034`, `RFC 1035`
Synopsis:

```c
#include <stdlib.h>

double drand48(void);
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `drand48()` function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative `double` uniformly distributed over the interval `[0.0, 1.0]`.

Call one of `lcong48()`, `seed48()`, or `srand48()` to initialize the random-number generator before calling `drand48()`, `lrand48()`, or `mrand48()`.

The `erand48()` function is a thread-safe version of `drand48()`.

Returns:

A pseudo-random `double`.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

\textit{erand48()}, \textit{jrand48()}, \textit{lcong48()}, \textit{lrand48()}, \textit{mrand48()}, \textit{nrand48()}, \textit{seed48()}, \textit{srand48()}
**drem(), dremf()**

*Compute the remainder of two numbers*

**Synopsis:**

```c
#include <math.h>

double drem ( double x, 
              double y );

float dremf ( float x, 
              float y );
```

**Arguments:**

- **x** The numerator of the division.
- **y** The denominator.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `drem()` and `dremf()` functions compute the remainder \( r = x - n \times y \), when \( y \) is nonzero. The value \( n \) is the integral value nearest the exact value \( x/y \).

When \( | n - x/y | = \frac{1}{2} \), the value \( n \) is chosen to be even. But `remainder(x, 0)` and `remainder(infinity,0)` are invalid operations that produce a NaN.

The behavior of `drem()` is independent of the rounding mode.

**Returns:**

The remainder, \( r = x - n \times y \), when \( y \) is nonzero.
Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

remainder()


**ds_clear()**

Delete a data server variable

**Synopsis:**

```c
#include <ds.h>

int ds_clear( ds_t dsdes,
              const char* variable_name );
```

**Arguments:**

- `dsdes` A data server descriptor returned by `ds_register()`.
- `variable_name` The name of the variable that you want to delete.

**Library:**

`libds`

Use the `-l ds` option to `qcc` to link against this library.

**Description:**

The `ds_clear()` function deletes `variable_name` from the data server identified by `dsdes`.

**Returns:**

- 0 Success.
- -1 An error occurred (`errno` is set).

**Errors:**

- EBADF Invalid file descriptor `dsdes`.
- ESRCH The variable doesn’t exist in the data server.
Examples:

See `slinger` in the *Utilities Reference*.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`ds_deregister()`, `ds_flags()`
Create a data server variable

Synopsis:

```c
#include <ds.h>

int ds_create( ds_t dsdes,
               const char * variable_name,
               char flags,
               struct sigevent * sigevent );
```

Arguments:

- **dsdes** A data server descriptor returned by `ds_register()`.
- **variable_name** The name of the variable that you want to create.
  All variables are global, so only one instance of the variable can exist in the data server process. The maximum length of a variable name is 60 characters.
- **flags** Flags that specify the variable’s behavior:
  - DS_PERM — don’t delete the variable when the application that created it terminates. The variable is removed when the data server process terminates, or if the flag is turned off after the application that created the variable terminates.
  - If `flags` is 0, the variable is removed if you call `ds_deregister()`, or the process terminates.
- **sigevent** A pointer to a `sigevent` structure that describes a proxy or signal to be sent to the external application that created the variable if the data referenced by the variable changes; see below.

Library:

- `libds`

Use the `-l ds` option to `gcc` to link against this library.
ds_create()

Description:

The `ds_create()` function creates a variable, whose name is given by `variable_name`, on the data server identified by `dsdes`.

If the data referenced by `variable_name` changes, a proxy or signal, described in the `sigevent` structure, can be sent to the external application that created `variable_name` (see `ds_set()`).

We recommend the following event types for use with this function:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE
- SIGEV_INTR

To display the current value of a variable on an HTML page, use the `qnxvar` token with the `read` tag. See the description of `slinger` in the Utilities Reference.

Returns:

0   Success.
-1   An error occurred (`errno` is set).

Errors:

EBADF   Invalid file descriptor `dsdes`.
EEXIST   The variable name already exists in the data server.
ENOMEM   Not enough memory to create the variable or initialize the data.
Examples:

See `slinger` in the *Utilities Reference*.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`ds_flags()`, `ds_get()`, `ds_register()`, `ds_set()`, `sigevent`
Synopsis:

```c
#include <ds.h>

int ds_deregister( ds_t dsdes );
```

Arguments:

- `dsdes`: A data server descriptor returned by `ds_register()`.

Library:

- `libds`

Use the `-l ds` option to `qcc` to link against this library.

Description:

The `ds_deregister()` function deregisters your application with the data server, `dsdes`, and deletes any variables that the data server process created, except those with the DS_PERM flag set (see `ds_flags()`).

Returns:

- `0`: Success.
- `-1`: An error occurred (`errno` is set).

Errors:

- `EBADF`: Invalid file descriptor, `dsdes`.

Examples:

See `slinger` in the Utilities Reference.
ds_deregister()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ds_flags(), ds_register()
**ds_flags()**

*Set the flags for a data server variable*

**Synopsis:**

```c
#include <ds.h>

int ds_flags( ds_t dsdes,
              const char* variable_name,
              char flags );
```

**Arguments:**

- `dsdes`: A data server descriptor returned by `ds_register()`.
- `variable_name`: The name of the data server variable.
- `flags`: The new flags for the variable. The flags include:
  - `DS_PERM`: Don’t delete the variable when the application that created it terminates. The variable is removed when the data server process terminates, or if the flag is turned off after the application that created the variable terminates.

**Library:**

- `libds`
  
  Use the `-l ds` option to `qcc` to link against this library.

**Description:**

The `ds_flags()` function changes the state of the flags belonging to the variable called `variable_name` on the data server identified by `dsdes`.

**Returns:**

- `0` for success, or `-1` if an error occurs (`errno` is set).
Errors:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid file descriptor \textit{dsdes}.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The variable doesn’t exist in the data server.</td>
</tr>
</tbody>
</table>

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

\textit{ds\_clear()}, \textit{ds\_create()}, \textit{ds\_deregister()}, \textit{ds\_set()}
ds_get()

Retrieve a data server variable

Synopsis:

```
#include <ds.h>

int ds_get( ds_t dsdes,
            const char* variable_name,
            const char* variable_data,
            size_t data_len );
```

Arguments:

- `dsdes`: A data server descriptor returned by `ds_register()`.
- `variable_name`: The name of the data server variable that you want to get.
- `variable_data`: A buffer where the function can store the data associated with the variable.
- `data_len`: The size of the buffer, in bytes.

Library:

```
libds
```

Use the `-l ds` option to `qcc` to link against this library.

Description:

The `ds_get()` function retrieves the data corresponding to `variable_name` from the data server `dsdes`, and places it in the buffer pointed to by `variable_data`.

Returns:

The amount of data written to the buffer `variable_data`, or -1 if an error occurs (`errno` is set).
ds_get()

Errors:

EBADF    Invalid file descriptor dsdes.
EMSGSIZE The buffer isn’t big enough for the data.
ESRCH    The variable doesn’t exist in the data server.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ds_create(), ds_set()
**ds_register()**

Register an application with the data server

**Synopsis:**

```c
#include <ds.h>

ds_t ds_register( void );
```

**Library:**

```c
libds
```

Use the `-l ds` option to `qcc` to link against this library.

**Description:**

The `ds_register()` function registers your application with the data server. The data server must reside on the same node as your application.

**Returns:**

A data server descriptor, or -1 if an error occurs (`errno` is set).

**Errors:**

- `ENOENT` No such file or directory; the data server isn’t started.
- `ENOMEM` Insufficient memory is available to communicate with the data server.

**Examples:**

See `slinger` in the *Utilities Reference*.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
</table>

*continued…*
### ds_register()

#### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### See also:

```
    ds_deregister()
```
**ds_set()**

Set a data server variable

**Synopsis:**

```c
#include <ds.h>

int ds_set( ds_t dsdes,
            const char* variable_name,
            const char* variable_data,
            size_t data_len );
```

**Arguments:**

- **dsdes**
  A data server descriptor returned by `ds_register()`.

- **variable_name**
  The name of the data server variable that you want to set.

- **variable_data**
  A pointer to the data you want to associate with the variable.

- **data_len**
  The size of the data, in bytes.

**Library:**

```c
libds
```

Use the `-l ds` option to `qcc` to link against this library.

**Description:**

The `ds_set()` function passes the data `variable_data` to the data server identified by `dsdes`. The data server stores the data in the variable whose name is given by `variable_name`, overwriting any existing value.

To display the modified data on an HTML page, use the `qnxvar` token with the `read` tag. See the description of `slinger` in the Utilities Reference.
ds_set()

Returns:

0 for success, or -1 if an error occurs (errno is set).

Errors:

EBADF Invalid file descriptor dsdes.
ENOMEM Not enough memory to store the data.
ESRCH The variable doesn’t exist in the data server.

Examples:

See slinger in the Utilities Reference.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

d_create(), ds_flags(), ds_get()
Synopsis:

```c
#include <unistd.h>

int dup( int filedes );
```

Arguments:

- `filedes`: The file descriptor that you want to duplicate.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `dup()` function duplicates the file descriptor specified by `filedes`. The new file descriptor refers to the same open file descriptor as the original, and shares any locks. The new file descriptor also:

- references the same file or device
- has the same open mode (read and/or write)
- has an identical file position to the original (changing the position with one descriptor results in a changed position in the other).

Changing the file position with one descriptor results in a changed position for the other.

Calling:

```c
dup_filedes = dup( filedes );
```

is the same as:

```c
dup_filedes = fcntl( filedes, F_DUPFD, 0 );
```
Returns:

The new file descriptor for success, or -1 if an error occurs (errno is set).

Errors:

EBADF The file descriptor, filedes, isn’t a valid.
EMFILE There are already OPEN_MAX file descriptors in use.
ENOSYS The dup() function isn’t implemented for the filesystem specified by filedes.

Examples:

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    int filedes, dup_filedes;

    filedes = open( "file",
                    O_WRONLY | O_CREAT | O_TRUNC,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);

    if( filedes != -1 ) {
        dup_filedes = dup( filedes );
        if( dup_filedes != -1 ) {
            /* process file */
            /* ... */

            close( dup_filedes );
        }
        close( filedes );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chsize(), close(), creat(), dup2(), eof(), errno, execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(), fcntl(), fileno(), fstat(), isatty(), lseek(), open(), read(), sopen(), stat(), tell(), umask(), write()
**dup2()**

Duplicate a file descriptor, specifying the new descriptor

### Synopsis:

```c
#include <unistd.h>

int dup2( int filedes,
          int filedes2 );
```

### Arguments:

- **filedes** The file descriptor that you want to duplicate.
- **filedes2** The number that you want to use for the new file descriptor.

### Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `dup2()` function duplicates the file descriptor specified by `filedes`. The number of the new file descriptor will be `filedes2`. If a file already is opened with this descriptor, the file is closed before the duplication is attempted.

The new file descriptor:

- references the same file or device
- has the same open mode (read and/or write)
- has an identical file position to the original (changing the position with one descriptor results in a changed position in the other).

Calling:

```c
dup_filedes = dup2( filedes, filedes2 );
```

Is the same as:

```c
close( filedes2 );
dup_filedes = fcntl( filedes , F_DUPFD, filedes2 );
```
Returns:

The value of `filedes2` for success, or -1 if an error occurs (`errno` is set).

Errors:

- **EBADF** The file descriptor, `filedes` isn’t a valid open file descriptor, or `filedes2` is out of range.
- **EMFILE** There are already OPEN_MAX file descriptors in use.
- **ENOSYS** The `dup2()` function isn’t implemented for the filesystem specified by `filedes`.

Examples:

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    int filedes , dup_filedes ;

    filedes = open( "file", 
        O_WRONLY | O_CREAT | O_TRUNC, 
        S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    if( filedes != -1 ) {
        dup_filedes = 4;
        if( dup2( filedes, dup_filedes ) != -1 ) {
            /* process file */
            /* ... */
            close( dup_filedes );
        } 
        close( filedes );
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```
**dup2()**

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`chsize()`, `close()`, `creat()`, `dup()`, `eof()`, `errno`, `execl()`, `execlp()`, `execle()`, `execvp()`, `execve()`, `fcntl()`, `fileno()`, `fstat()`, `isatty()`, `lseek()`, `open()`, `read()`, `sopen()`, `stat()`, `tell()`, `umask()`, `write()`
eaccess()

Check to see if a file or directory can be accessed (extended version)

Synopsis:

```
#include <libgen.h>
#include <unistd.h>

int eaccess( const char * path, int amode );
```

Arguments:

- `path` The path to the file or directory that you want to access.
- `amode` The access mode you want to check. This must be either:
  - `F_OK` — test for file existence.
  - `R_OK` — test for read permission.
  - `W_OK` — test for write permission.
  - `X_OK` — for a directory, test for search permission. Otherwise, test for execute permission.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `eaccess()` function is an extended version of `access()`. It checks if the file or directory specified by `path` exists and if it can be accessed with the file access permissions given by `amode`. However, unlike `access()`, it uses the effective user ID and effective group ID.
`eaccess()`

Returns:

- 0  The file or directory exists and can be accessed with the specified mode.

- -1 An error occurred (errno is set.)

Errors:

- `EACCES`  The permissions specified by `amode` are denied, or search permission is denied on a component of the path prefix.

- `EINVAL`  An invalid value was specified for `amode`.

- `ELOOP`  Too many levels of symbolic links or prefixes.

- `ENAMETOOLONG`  The length of the path string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

- `ENOENT`  A component of the path isn’t valid.

- `ENOSYS`  The `eaccess()` function isn’t implemented for the filesystem specified in `path`.

- `ENOTDIR`  A component of the path prefix isn’t a directory.

- `EROFS`  Write access was requested for a file residing on a read-only file system.

Classification:

Unix

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

Continued…
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`access()`, `chmod()`, `errno`, `fstat()`, `open()`, `stat()`
Synopsis:

N/A

Description:

This linker symbol defines the end of the data segment, excluding BSS data. This variable isn’t defined in any header file.

Classification:

QNX Neutrino

See also:

brk(), _btext, _end, _etext, sbrk()
Synopsis:

```c
#include <unistd.h>

void encrypt( char block[64],
              int flag );
```

Arguments:

- `block`: A 64-character array of binary values to encrypt. The function stores the encrypted value in the same array.
- `flag`: If the value of `flag` is 0, the function encrypts `block`; otherwise, `encrypt()` fails.

Library:

- `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

  This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `encrypt()` function uses the NBS Data Encryption Standard (DES) algorithm and the key you specify by calling `setkey()` to encrypt the given block of data.

Classification:

- POSIX 1003.1 XSI

Safety

- Cancellation point: No

  
  `continued...`
encrypt()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

crypt(), setkey()
Synopsis:

N/A

Description:

This linker symbol defines the end of the data segment, including BSS data. This variable isn’t defined in any header file.

Classification:

QNX Neutrino

See also:

brk(), _btext, _edata, _etext, sbrk()
endgrent()

Close the group database file

Synopsis:

```c
#include <grp.h>

int endgrent( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `endgrent()` routine closes the group name database file, so all group access routines behave as if `setgrent()` had never been called.

Returns:

Zero.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`getgrent()`, `setgrent()`
endhostent()

Close the TCP connection and the hosts file

Synopsis:

```
#include <netdb.h>

void endhostent( void );
```

Library:

```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `endhostent()` routine closes the TCP connection and the hosts file.

Files:

```
/etc/hosts   Host database file.
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`gethostbyaddr()`, `gethostbyname()`, `gethostent()`, `hostent()`, `sethostent()`

`/etc/hosts`, `/etc/resolv.conf` in the `Utilities Reference`
**Synopsis:**

```
#include <gulliver.h>

uint16_t ENDIAN_BE16( uint16_t num );
```

**Arguments:**

- `num`  The big-endian number you want to convert.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ENDIAN_BE16()` macro returns the native version of the big-endian value `num`.

**Returns:**

The native-endian value of `num`.

**Examples:**

```
Convert a big-endian value to native-endian:

#include <stdio.h>  
#include <stdlib.h>  
#include <gulliver.h>  
#include <inttypes.h>  

int main( void )
{
    uint16_t val = 0x1234;
    printf( "0x%04x = 0x%04x\n",
            val, ENDIAN_BE16( val ) );
    return EXIT_SUCCESS;
}
```
On a little-endian system, this prints:
0x1234 = 0x3412

Classification:
QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

`ENDIAN_BE16()` is implemented as a macro.

See also:

`ENDIAN_BE32()`, `ENDIAN_BE64()`, `ENDIAN_LE16()`, `ENDIAN_LE32()`, `ENDIAN_LE64()`, `ENDIAN_RET16()`, `ENDIAN_RET32()`, `ENDIAN_RET64()`, `ENDIAN_SWAP16()`, `ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`, `ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`, `UNALIGNED_RET64()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
Return a big-endian 32-bit value in native format

Synopsis:
#include <gulliver.h>

uint32_t ENDIAN_BE32( uint32_t num );

Arguments:

num The big-endian number you want to convert.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The ENDIAN_BE32() macro returns the native version of the big-endian value num.

Returns:

The native-endian value of num.

Examples:

Convert a big-endian value to native-endian:

#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;

    printf( "0x%08x = 0x%08x\n", 
            val, ENDIAN_BE32( val ) );

    return EXIT_SUCCESS;
}
On a little-endian system, this prints:
0xdeadbeef = 0xefbeadde

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

`ENDIAN_BE32()` is implemented as a macro.

See also:

`ENDIAN_BE16()`, `ENDIAN_BE64()`, `ENDIAN_LE16()`, 
`ENDIAN_LE32()`, `ENDIAN_LE64()`, `ENDIAN_RET16()`, 
`ENDIAN_RET32()`, `ENDIAN_RET64()`, `ENDIAN_SWAP16()`, 
`ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`, 
`ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`, 
`UNALIGNED_RET64()`, `UNALIGNED_PUT16()`, 
`UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
SYNOPSIS:

#include <gulliver.h>

uint64_t ENDIAN_BE64( uint64_t num );

ARGUMENTS:

num The big-endian number you want to convert.

LIBRARY:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

DESCRIPTION:

The ENDIAN_BE64() macro returns the native version of the big-endian value num.

RETURNS:

The native-endian value of num.

EXAMPLES:

Convert a big-endian value to native-endian:

#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678;

    printf( "0x%016llx = 0x%016llx\n", val, ENDIAN_BE64( val ) );

    return EXIT_SUCCESS;
}
On a little-endian system, this prints:

\[ 0x1234deadbeef5678 = 0x7856efbeadde3412 \]

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: Yes
- Signal handler: Yes
- Thread: Yes

**Caveats:**

`ENDIAN_BE64()` is implemented as a macro.

**See also:**

- `ENDIAN_BE16()`, `ENDIAN_BE32()`, `ENDIAN_LE16()`, `ENDIAN_LE32()`, `ENDIAN_LE64()`, `ENDIAN_RET16()`, `ENDIAN_RET32()`, `ENDIAN_RET64()`, `ENDIAN_SWAP16()`, `ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`, `ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`, `UNALIGNED_RET64()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
ENDIAN LE16()  © 2007, QNX Software Systems GmbH & Co. KG.

Return a little-endian 16-bit value in native format

Synopsis:

```c
#include <gulliver.h>

uint16_t ENDIAN_LE16( uint16_t num );
```

Arguments:

- `num` The little-endian number you want to convert.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ENDIAN_LE16()` macro returns the native version of the little-endian value `num`.

Returns:

The native-endian value of `num`.

Examples:

Convert a little-endian value to native-endian:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;
    printf( "0x%04x = 0x%04x\n", val, ENDIAN_LE16( val ) );
    return EXIT_SUCCESS;
}
```
On a big-endian system, this prints:

\[ 0x1234 = 0x3412 \]

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

`ENDIAN_LE16()` is implemented as a macro.

**See also:**

`ENDIAN_BE16()`, `ENDIAN_BE32()`, `ENDIAN_BE64()`, `ENDIAN_LE32()`, `ENDIAN_LE64()`, `ENDIAN_RET16()`, `ENDIAN_RET32()`, `ENDIAN_RET64()`, `ENDIAN_SWAP16()`, `ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`, `ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`, `UNALIGNED_RET64()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
Return a little-endian 32-bit value in native format

Synopsis:
```c
#include <gulliver.h>

uint32_t ENDIAN_LE32( uint32_t num );
```

Arguments:
- `num` The little-endian number you want to convert.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `ENDIAN_LE32()` macro returns the native version of the little-endian value `num`.

Returns:
The native-endian value of `num`.

Examples:
Convert a little-endian value to native-endian:
```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;

    printf( "0x%08x = 0x%08x\n",
            val, ENDIAN_LE32( val ) );

    return EXIT_SUCCESS;
}
```
On a big-endian system, this prints:

```
0xdeadbeef = 0xefbeadde
```

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

`ENDIAN_LE32()` is implemented as a macro.

**See also:**

`ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET16(), ENDIAN_RET32(), ENDIAN_RET64(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohl(),
ntobs(), UNALIGNED_RET16(), UNALIGNED_RET32(),
UNALIGNED_RET64(), UNALIGNED_PUT16(),
UNALIGNED_PUT32(), UNALIGNED_PUT64()`
Return a little-endian 64-bit value in native format

Synopsis:
```
#include <gulliver.h>

uint64_t ENDIAN_LE64( uint64_t num );
```

Arguments:

- `num` - The little-endian number you want to convert.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ENDIAN_LE64()` macro returns the native version of the little-endian value `num`.

Returns:

The native-endian value of `num`.

Examples:

Convert a little-endian value to native-endian:
```
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678;

    printf( "%016llx = %016llx\n", val, ENDIAN_LE64( val ) );

    return EXIT_SUCCESS;
}
```
On a big-endian system, this prints:
0x1234deadbeef5678 = 0x7856efbeadde3412

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

`ENDIAN_LE64()` is implemented as a macro.

See also:

`ENDIAN_BE16()`, `ENDIAN_BE32()`, `ENDIAN_BE64()`,
`ENDIAN_LE16()`, `ENDIAN_LE32()`, `ENDIAN_RET16()`,
`ENDIAN_RET32()`, `ENDIAN_RET64()`, `ENDIAN_SWAP16()`,
`ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`,
`ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`,
`UNALIGNED_RET64()`, `UNALIGNED_PUT16()`,
`UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
**ENDIAN_RET16()**

Return an endian-swapped 16-bit value

**Synopsis:**

```c
#include <gulliver.h>

uint16_t ENDIAN_RET16( uint16_t num );
```

**Arguments:**

- `num`: The number you want to convert.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ENDIAN_RET16()` macro returns the endian-swapped value of `num`.

**Returns:**

The endian-swapped value of `num`.

**Examples:**

Swap the endianness of a value:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;
    printf( "0x%04x = 0x%04x\n",
            val, ENDIAN_RET16( val ) );
    return EXIT_SUCCESS;
}
```
This prints:
0x1234 = 0x3412

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

ENDIAN_RET16() is implemented as a macro.

See also:

ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET32(), ENDIAN_RET64(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohl(),
ntohs(), UNALIGNED_RET16(), UNALIGNED_RET32(),
UNALIGNED_RET64(), UNALIGNED_PUT16(),
UNALIGNED_PUT32(), UNALIGNED_PUT64()
**ENDIAN_RET32()**

Return an endian-swapped 32-bit value

**Synopsis:**

```c
#include <gulliver.h>

uint32_t ENDIAN_RET32( uint32_t num );
```

**Arguments:**

- `num` The number you want to convert.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ENDIAN_RET32()` macro returns the endian-swapped value of `num`.

**Returns:**

The endian-swapped value of `num`.

**Examples:**

Swap the endianness of a value:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;
    printf( "0x%x = 0x%08x \n", val, ENDIAN_RET32( val ) );
    return EXIT_SUCCESS;
}
```
This prints:
\texttt{\texttt{0xdeadbeef} = \texttt{0xefbeadde}}

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

\texttt{ENDIAN\_RET32()} is implemented as a macro.

**See also:**

\texttt{ENDIAN\_BE16()}, \texttt{ENDIAN\_BE32()}, \texttt{ENDIAN\_BE64()}, \texttt{ENDIAN\_LE16()}, \texttt{ENDIAN\_LE32()}, \texttt{ENDIAN\_LE64()}, \texttt{ENDIAN\_RET16()}, \texttt{ENDIAN\_RET64()}, \texttt{ENDIAN\_SWAP16()}, \texttt{ENDIAN\_SWAP32()}, \texttt{ENDIAN\_SWAP64()}, \texttt{htonl()}, \texttt{htons()}, \texttt{ntohl()}, \texttt{ntohs()}, \texttt{UNALIGNED\_RET16()}, \texttt{UNALIGNED\_RET32()}, \texttt{UNALIGNED\_RET64()}, \texttt{UNALIGNED\_PUT16()}, \texttt{UNALIGNED\_PUT32()}, \texttt{UNALIGNED\_PUT64()}
**ENDIAN RET64()**

Return an endian-swapped 64-bit value

**Synopsis:**

```c
#include <gulliver.h>

uint64_t ENDIAN_RET64( uint64_t num );
```

**Arguments:**

`num` The number you want to convert.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ENDIAN_RET64()` macro returns the endian-swapped value of `num`.

**Returns:**

The endian-swapped value of `num`.

**Examples:**

Swap the endianness of a value:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678;
    printf( "0x%016llx = 0x%016llx\n",
            val, ENDIAN_RET64( val ) );

    return EXIT_SUCCESS;
}
```
This prints:
0x1234deadbeef5678 = 0x7856efbeadde3412

Classification:
QNX Neutrino

Safety
- Cancellation point: No
- Interrupt handler: Yes
- Signal handler: Yes
- Thread: Yes

Caveats:
ENDIAN_RET64() is implemented as a macro.

See also:
ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET16(), ENDIAN_RET32(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohl(),
ntobs(), UNALIGNED_RET16(), UNALIGNED_RET32(),
UNALIGNED_RET64(), UNALIGNED_PUT16(),
UNALIGNED_PUT32(), UNALIGNED_PUT64()
**ENDIAN_SWAP16()**

Endian-swap a 16-bit value in place

**Synopsis:**

```c
#include <gulliver.h>

void ENDIAN_SWAP16( uint16_t * num );
```

**Arguments:**

num A pointer to the number you want to convert.

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ENDIAN_SWAP16()` macro endian-swaps the value pointed to by `num` in place.

**Examples:**

Swap the endianness of a value:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint16_t val = 0x1234;
    ENDIAN_SWAP16( &val );

    printf( "val = 0x%04x\n", val );
    return EXIT_SUCCESS;
}
```

This prints:

```c
val = 0x3412
```

540 QNX Neutrino Functions and Macros
**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

`ENDIAN_SWAP16()` is implemented as a macro.

**See also:**

`ENDIAN_BE16()`, `ENDIAN_BE32()`, `ENDIAN_BE64()`,
`ENDIAN_LE16()`, `ENDIAN_LE32()`, `ENDIAN_LE64()`,
`ENDIAN_RET16()`, `ENDIAN_RET32()`, `ENDIAN_RET64()`,
`ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`,
`ntobs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`,
`UNALIGNED_RET64()`, `UNALIGNED_PUT16()`,
`UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
**ENDIAN_SWAP32()**  © 2007, QNX Software Systems GmbH & Co. KG.

Endian-swap a 32-bit value in place

**Synopsis:**

```c
#include <gulliver.h>

void ENDIAN_SWAP32( uint32_t * num );
```

**Arguments:**

\(num\)  
A pointer to the number you want to convert.

**Library:**

libc

Use the \(-l c\) option to \(qcc\) to link against this library. This library is usually included automatically.

**Description:**

The **ENDIAN_SWAP32()** macro endian-swaps the value pointed to by \(num\) in place.

**Examples:**

Swap the endianness of a value:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint32_t val = 0xdeadbeef;
    ENDIAN_SWAP32( &val );
    printf( "val = 0x%08x\n", val );
    return EXIT_SUCCESS;
}
```

This prints:

\(val = 0xefbeadde\)
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

(ENDIAN_SWAP32()) is implemented as a macro.

See also:

(ENDIAN_BE16()), (ENDIAN_BE32()), (ENDIAN_BE64()),
(ENDIAN_LE16()), (ENDIAN_LE32()), (ENDIAN_LE64()),
(ENDIAN RET16()), (ENDIAN RET32()), (ENDIAN RET64()),
(ENDIAN_SWAP16()), (ENDIAN_SWAP32()), (ENDIAN_SWAP64()),
(hton1()), (htons()), (ntohl()),
(ntohs()), (UNALIGNED RET16()), (UNALIGNED RET32()),
(UNALIGNED RET64()), (UNALIGNED PUT16()),
(UNALIGNED PUT32()), (UNALIGNED PUT64())
**ENDIAN_SWAP64()**

Endian-swap a 64-bit value in place

**Synopsis:**

```c
#include <gulliver.h>

void ENDIAN_SWAP64( uint64_t * num );
```

**Arguments:**

- `num` A pointer to the number you want to convert.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ENDIAN_SWAP64()` macro endian-swaps the value pointed to by `num` in place.

**Examples:**

Swap the endianness of a value:

```c
#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678LL;
    ENDIAN_SWAP16( &val );

    printf( "val = 0x%016lx\n", val );

    return EXIT_SUCCESS;
}
```

This prints:

`val = 0x7856efbeadde3412`

© 2007, QNX Software Systems GmbH & Co. KG.
**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

*ENDIAN_SWAP64()* is implemented as a macro.

**See also:**

*ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(), ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(), ENDIAN RET16(), ENDIAN RET32(), ENDIAN RET64(), ENDIAN_SWAP16(), ENDIAN_SWAP32(), htonl(), htons(), ntohl(), ntohs(), UNALIGNED_PUT16(), UNALIGNED_PUT32(), UNALIGNED PUT64(), UNALIGNED RET16(), UNALIGNED RET32(), UNALIGNED RET64(),
**Synopsis:**

```c
#include <netdb.h>

void endnetent( void );
```

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `endnetent()` routine closes the network name database file.

**Files:**

`/etc/networks`

Network name database file.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`getnetbyaddr()`, `getbyname()`, `getnetent()`, `netent`, `setnetent()`

`/etc/networks` in the *Utilities Reference*
endprotoent()

Close the protocol name database file

Synopsis:

```c
#include <netdb.h>

void endprotoent( void );
```

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `endprotoent()` routine closes the protocol name database file.

Files:

`/etc/protocols`

Protocol name database file.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`getprotobyname()`, `getprotobynumber()`, `getprotoent()`, `protoent()`, `setprotoent()`

`/etc/protocols` in the *Utilities Reference*
**Synopsis:**
```
#include <sys/types.h>
#include <pwd.h>

int endpwent( void );
```

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `endpwent()` function closes the password name database file, so all password access routines behave as if `setpwent()` had never been called.

**Returns:**
Zero.

**Classification:**
POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

getpwent(), setpwent()
endservent()

Close the network services database file

Synopsis:

```c
#include <netdb.h>

void endservent( void );
```

Library:

```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `endservent()` routine closes the network services database file.

Files:

```
/etc/services
```

Network services database file.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`getservbyname(), getservbyport(), getservent(), servent, setservent()`

`/etc/services` in the Utilities Reference
endspent()  
Close the shadow password database file

Synopsis:
#include <sys/types.h>
#include <shadow.h>

    void endspent( void );

Library:

    libc

    Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

    The endspent() function closes the shadow password database file, so all password access routines behave as if setspent() had never been called.

Returns:

    Zero.

Classification:

    Unix

    | Safety                  |
    |-------------------------|
    | Cancellation point   Yes|
    | Interrupt handler     No|
    | Signal handler        No|
    | Thread                 No|
See also:

getspent(), setspent()
Synopsis:

```
#include <utmp.h>

void endutent( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `endutent()` function closes the currently open file specified in `_PATH_UTMP`.

Files:

```
_PATH_UTMP
```

The name of the user information file.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
endutent()

See also:

getutent(), getutid(), getutline(), pututline(), setutent(), utmp, utmpname()

login in the Utilities Reference
environ

Pointer to the process’s environment variables

Synopsis:

```c
#include <unistd.h>

extern char ** environ;
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

When a process begins, an array of strings called the `environment` is made available. This array is pointed to by the external variable `environ`. The strings in the array have the form:

`variable=value`

and are terminated by a NULL pointer.

Classification:

POSIX 1003.1

Caveats:

Don’t modify `environ` directly; use `clearenv()`, `getenv()`, `putenv()`, `searchenv()`, `setenv()`, and `unsetenv()`.

Changes to the environment affect all threads in a multithreaded process.

See also:

`clearenv()`, `getenv()`, `putenv()`, `searchenv()`, `setenv()`, `unsetenv()`
Test if the end-of-file has been reached

Synopsis:

```c
#include <unistd.h>

int eof( int filedes );
```

Arguments:

`filedes` A file descriptor for the file that you want to check.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `eof()` function is a low-level function that determines if the end of the file specified by `filedes` has been reached.

Input operations set the current file position; you can call the `eof()` function to detect the end of the file before more input operations to prevent attempts at reading beyond the end of the file.

Returns:

1 The end-of-file has been reached.
0 The end-of-file hasn’t been reached.
-1 An error occurred (`errno` is set).

Errors:

`EBADF` The file descriptor, `filedes`, isn’t valid.
Examples:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes , len;
    char buffer[100];

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        while( ! eof( filedes ) ) {
            len = read( filedes , buffer, sizeof(buffer) - 1 );
            buffer[ len ] = '\0';
            printf( "%s", buffer );
        }
        close( filedes );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th>Yes</th>
<th>No</th>
<th>Yes</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See also:

errno, feof(), open(), read()
**erand48()**

*Generate a pseudo-random* double *in a thread-safe manner*

**Synopsis:**

```c
#include <stdlib.h>

double erand48( unsigned short int xsubi[3] );
```

**Arguments:**

*xsubi*  An array that comprises the 48 bits of the initial value that you want to use.

**Library:**

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

**Description:**

The *erand48()* function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative *double* uniformly distributed over the interval [0.0, 1.0]. It’s a thread-safe version of *drand48()*.

The *xsubi* array should contain the desired initial value; this makes *erand48()* thread-safe, and lets you start a sequence of random numbers at any known value.

**Returns:**

A pseudo-random *double*.

**Classification:**

POSIX 1003.1 XSI

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>continued...</td>
<td></td>
</tr>
</tbody>
</table>
erand48()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

drand48(), jrand48(), lcong48(), lrand48(), mrand48(), nrand48(),
seed48(), srand48()
erf(), erff()
Compute the error function of a number

Synopsis:

```c
#include <math.h>

double erf ( double x );
float erff ( float x );
```

Arguments:

- `x` The number for which you want to compute the error function.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `erf()` and `erff()` functions compute the following:

\[
\frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt
\]

If \( x \) is large and the result of `erf()` is subtracted from \( 1.0 \), the results aren’t very accurate; use `erfc()` instead.

This equality is true: \( erf(-x) = -erf(x) \)

Returns:

The value of the error function, or `NAN` if \( x \) is `NAN`. 
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`erfc()`
Synopsis:

```c
#include <math.h>

double erfc ( double x );

float erfcf ( float x );
```

Arguments:

- `x` The number for which you want to compute the complementary error function.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `erfc()` and `erfcf()` functions calculate the complementary error function of `x` (i.e. the result of the error function, `erf()`, subtracted from 1.0). This is useful because the error function isn’t very accurate when `x` is large.

The `erf()` function computes:

\[
\frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt
\]

This equality is true: `erfc(-x) = 2 - erfc(x)`

Returns:

The value of the error function, or `NAN` if `x` is `NAN`. 
erfc(), erfcf()

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set errno to 0, call the function, and then check errno again. These functions don’t change errno if no errors occurred.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

erf()
Synopsis:

```c
#include <err.h>

void err( int eval,
           const char *fmt, ...);

void errx( int eval,
           const char *fmt, ...);
```

Arguments:

- **eval**: The value to use as the exit code of the process.
- **fmt**: NULL, or a `printf()`-style string used to format the message.
- Additional arguments
  - As required by the format string.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `err()` and `warn()` family of functions display a formatted error message on `stderr`:

- The functions without an `x` in their names display the string associated with the current value of `errno`.
- Those with a `v` are “varargs” functions.
- Those with `err` exit instead of returning.
The `err()` function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the `fmt` argument isn’t NULL
- the string associated with the current value of `errno`
- a newline character.

The `errx()` function produces a similar message, except that it doesn’t include the string associated with `errno`. The message consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, if the `fmt` argument isn’t NULL
- a newline character.

The `err()` and `errx()` functions don’t return, but exit with the value of the argument `eval`. 

<table>
<thead>
<tr>
<th>Function</th>
<th><code>errno</code> string?</th>
<th>Varargs?</th>
<th>Exits?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>err()</code></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>errx()</code></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>verr()</code></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>verrx()</code></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>vwarn()</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>vwarnx()</code></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>warn()</code></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>warnx()</code></td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Examples:

Display the current *errno* information string and exit:

```c
if ((p = malloc(size)) == NULL)
  err(1, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
  err(1, "%s", file_name);
```

Display an error message and exit:

```c
if (tm.tm_hour < START_TIME)
  errx(1, "too early, wait until %s", start_time_string);
```

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`printf()`, `stderr`, `strerror()`, `verr()`, `verrx()`, `vwarn()`, `vwarnx()`, `warn()`, `warnx()`
**errno**

*Global error variable*

**Synopsis:**

```c
#include <errno.h>

extern int errno;

char * const sys_errlist[];
int sys_nerr;
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `errno` variable is set to certain error values by many functions whenever an error has occurred.

---

You can’t assume that the value of `errno` is valid unless the function that you’ve called indicates that an error has occurred. The runtime library never resets `errno` to 0.

The documentation for a function might list special meanings for certain values of `errno`, but this doesn’t mean that these are the only values that the function might set.

---

The `errno` variable may be implemented as a macro, but you can always examine or set it as if it were a simple integer variable.

---

Each thread in a multi-threaded program has its own error value in its thread local storage. No matter which thread you’re in, you can simply refer to `errno` — it’s defined in such a way that it refers to the correct variable for the thread. For more information, see “Local storage for private data” in the documentation for `ThreadCreate()`.

---

The following variables are also defined in `<errno.h>`:
sys_errno list  An array of error messages corresponding to errno.

sys_nerr  The number of entries in the sys_errno list array.

The values for errno include at least the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>Argument list is too long</td>
</tr>
<tr>
<td>EACCES</td>
<td>Permission denied</td>
</tr>
<tr>
<td>EADDRINUSE</td>
<td>Address is already in use</td>
</tr>
<tr>
<td>EADDRNOTAVAIL</td>
<td>Can’t assign requested address</td>
</tr>
<tr>
<td>EADV</td>
<td>Advertise error</td>
</tr>
<tr>
<td>EAFNOSUPPORT</td>
<td>Address family isn’t supported by</td>
</tr>
<tr>
<td></td>
<td>protocol family</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Resource is temporarily unavailable; try</td>
</tr>
<tr>
<td></td>
<td>again</td>
</tr>
<tr>
<td>EALREADY</td>
<td>Operation is already in progress</td>
</tr>
<tr>
<td>EBADE</td>
<td>Invalid exchange</td>
</tr>
<tr>
<td>EBADF</td>
<td>Bad file descriptor</td>
</tr>
<tr>
<td>EBADFD</td>
<td>FD is invalid for this operation</td>
</tr>
<tr>
<td>EBADFSYS</td>
<td>Corrupted filesystem detected</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>Bad message (1003.1b-1993)</td>
</tr>
<tr>
<td>EBADR</td>
<td>Invalid request descriptor</td>
</tr>
<tr>
<td>EBADRPC</td>
<td>RPC struct is bad</td>
</tr>
<tr>
<td>EBADRQC</td>
<td>Invalid request code</td>
</tr>
<tr>
<td>EBADSLT</td>
<td>Invalid slot</td>
</tr>
<tr>
<td>EBFONT</td>
<td>Bad font-file format</td>
</tr>
</tbody>
</table>

continued...
### Value | Meaning
---|---
EBUSY | Device or resource is busy
ECANCELED | Operation canceled (1003.1b-1993)
ECHILD | No child processes
ECHRN | Channel number is out of range
ECOMM | Communication error occurred on send
ECOMNBORTED | Software caused connection to abort
ECOMREFUSED | Connection refused
ECOMRESET | Connection reset by peer
ECTRLTERM | Remap to the controlling terminal
EDEADLK | Resource deadlock avoided
EDEADLOCK | File locking deadlock
EDESTADDRREQ | Destination address is required
EDOM | Math argument is out of domain for the function
EDQUOT | Disk quota exceeded
EEXIST | File exists
EFAULT | Bad address
EFBIG | File is too large
HOSTDOWN | Host is down
HOSTUNREACH | Unable to communicate with remote node
IDRM | Identifier removed
ILSEQ | Illegal byte sequence
INPROGRESS | Operation now in progress

continued...
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINTR</td>
<td>Interrupted function call</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid argument</td>
</tr>
<tr>
<td>EIO</td>
<td>I/O error</td>
</tr>
<tr>
<td>EISCONN</td>
<td>Socket is already connected</td>
</tr>
<tr>
<td>EISDIR</td>
<td>Is a directory</td>
</tr>
<tr>
<td>EL2HLT</td>
<td>Level 2 halted</td>
</tr>
<tr>
<td>EL2NSYNC</td>
<td>Level 2 not synchronized</td>
</tr>
<tr>
<td>EL3HLT</td>
<td>Level 3 halted</td>
</tr>
<tr>
<td>EL3RST</td>
<td>Level 3 reset</td>
</tr>
<tr>
<td>ELIBACC</td>
<td>Can’t access shared library</td>
</tr>
<tr>
<td>ELIBBAD</td>
<td>Accessing a corrupted shared library</td>
</tr>
<tr>
<td>ELIBEXEC</td>
<td>Attempting to exec a shared library</td>
</tr>
<tr>
<td>ELIBMAX</td>
<td>Attempting to link in too many libraries</td>
</tr>
<tr>
<td>ELIBSCN</td>
<td>The <code>.lib</code> section in <code>a.out</code> is corrupted</td>
</tr>
<tr>
<td>ELNRNG</td>
<td>Link number is out of range</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many levels of symbolic links or prefixes</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many open files</td>
</tr>
<tr>
<td>EMLINK</td>
<td>Too many links</td>
</tr>
<tr>
<td>EMORE</td>
<td>More to do, send message again</td>
</tr>
<tr>
<td>EMSGSIZE</td>
<td>Inappropriate message buffer length</td>
</tr>
<tr>
<td>EMULTIHOP</td>
<td>Multihop attempted</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>Filename is too long</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENETDOWN</td>
<td>Network is down</td>
</tr>
<tr>
<td>ENETRESET</td>
<td>Network dropped connection on reset</td>
</tr>
<tr>
<td>ENETUNREACH</td>
<td>Network is unreachable</td>
</tr>
<tr>
<td>ENFILE</td>
<td>Too many open files in the system</td>
</tr>
<tr>
<td>ENOANO</td>
<td>No anode</td>
</tr>
<tr>
<td>ENOBUFFS</td>
<td>No buffer space available</td>
</tr>
<tr>
<td>ENOCSI</td>
<td>No CSI structure available</td>
</tr>
<tr>
<td>ENODATA</td>
<td>No data (for no-delay I/O)</td>
</tr>
<tr>
<td>ENODEV</td>
<td>No such device</td>
</tr>
<tr>
<td>ENOENT</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>Exec format error</td>
</tr>
<tr>
<td>ENOLCK</td>
<td>No locks available</td>
</tr>
<tr>
<td>ENOLIC</td>
<td>No license available</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The link has been severed</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough memory</td>
</tr>
<tr>
<td>ENOMSG</td>
<td>No message of desired type</td>
</tr>
<tr>
<td>ENONDP</td>
<td>Need an NDP (8087...) to run</td>
</tr>
<tr>
<td>ENONET</td>
<td>Machine isn’t on the network</td>
</tr>
<tr>
<td>ENOPKG</td>
<td>Package isn’t installed</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>Protocol isn’t available</td>
</tr>
<tr>
<td>ENOREMOTE</td>
<td>Must be done on local machine</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>No space left on device</td>
</tr>
<tr>
<td>ENOSR</td>
<td>Out of streams resources</td>
</tr>
</tbody>
</table>

*continued…*
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOSTR</td>
<td>Device isn’t a stream</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>Function isn’t implemented</td>
</tr>
<tr>
<td>ENOTBLK</td>
<td>Block device is required</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>Socket isn’t connected</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>Not a directory</td>
</tr>
<tr>
<td>ENOTEMPTY</td>
<td>Directory isn’t empty</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>Socket operation on nonsocket</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>Not supported (1003.1b-1993)</td>
</tr>
<tr>
<td>ENOTTY</td>
<td>Inappropriate I/O control operation</td>
</tr>
<tr>
<td>ENOTUNIQ</td>
<td>Given name isn’t unique</td>
</tr>
<tr>
<td>ENXIO</td>
<td>No such device or address</td>
</tr>
<tr>
<td>EOK</td>
<td>No error</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>Operation isn’t supported</td>
</tr>
<tr>
<td>EOVERFLOW</td>
<td>Value too large to be stored in data type</td>
</tr>
<tr>
<td>EPERM</td>
<td>Operation isn’t permitted</td>
</tr>
<tr>
<td>EPFNOSUPPORT</td>
<td>Protocol family isn’t supported</td>
</tr>
<tr>
<td>EPIPE</td>
<td>Broken pipe</td>
</tr>
<tr>
<td>EPROCUNAVAIL</td>
<td>Bad procedure for program</td>
</tr>
<tr>
<td>EPROGMISMATCH</td>
<td>Program version wrong</td>
</tr>
<tr>
<td>EPROGUNAVAIL</td>
<td>RPC program isn’t available</td>
</tr>
<tr>
<td>EPROTO</td>
<td>Protocol error</td>
</tr>
<tr>
<td>EPROTONOSUPPORT</td>
<td>Protocol isn’t supported</td>
</tr>
<tr>
<td>EPROTOTYPE</td>
<td>Protocol is wrong type for socket</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERANGE</td>
<td>Result is too large</td>
</tr>
<tr>
<td>EREMCHG</td>
<td>Remote address changed</td>
</tr>
<tr>
<td>EREMOTE</td>
<td>The object is remote</td>
</tr>
<tr>
<td>ERESTART</td>
<td>Restartable system call</td>
</tr>
<tr>
<td>EROFS</td>
<td>Read-only filesystem</td>
</tr>
<tr>
<td>ERPCMISMATCH</td>
<td>RPC version is wrong</td>
</tr>
<tr>
<td>ESHUTDOWN</td>
<td>Can’t send after socket shutdown</td>
</tr>
<tr>
<td>ESOCKTNOSUPPORT</td>
<td>Socket type isn’t supported</td>
</tr>
<tr>
<td>ESPIPE</td>
<td>Illegal seek</td>
</tr>
<tr>
<td>ESRCH</td>
<td>No such process</td>
</tr>
<tr>
<td>ESRMNT</td>
<td>Server mount error</td>
</tr>
<tr>
<td>ESRVRFault</td>
<td>The receive side of a message transfer encountered a memory fault accessing the receive/reply buffer.</td>
</tr>
<tr>
<td>ESTALE</td>
<td>Potentially recoverable I/O error</td>
</tr>
<tr>
<td>ESTRPIPE</td>
<td>If pipe/FIFO, don’t sleep in stream head</td>
</tr>
<tr>
<td>ETIME</td>
<td>Timer expired</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>Connection timed out</td>
</tr>
<tr>
<td>ETOOMANYREFS</td>
<td>Too many references: can’t splice</td>
</tr>
<tr>
<td>ETXTBSY</td>
<td>Text file is busy</td>
</tr>
<tr>
<td>EUNATCH</td>
<td>Protocol driver isn’t attached</td>
</tr>
<tr>
<td>EUSERS</td>
<td>Too many users (for UFS)</td>
</tr>
<tr>
<td>EWOULDDBLOCK</td>
<td>Operation would block</td>
</tr>
<tr>
<td>EXDEV</td>
<td>Cross-device link</td>
</tr>
</tbody>
</table>

*continued...*
**errno**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXFULL</td>
<td>Exchange full</td>
</tr>
</tbody>
</table>

**Examples:**

```c
/*
 * The following program makes an illegal call
 * to the write() function, then prints the
 * value held in errno.
 */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main( void )
{
    int errvalue;
    errno = EOK;
    write( -1, "hello, world\n",
           strlen( "hello, world\n" ) );
    errvalue = errno;
    printf( "The error generated was %d\n", errvalue );
    printf( "That means: %s\n", strerror( errvalue ) );
}
```

**Classification:**

ANSI, POSIX 1003.1

**See also:**

`_errno`, `perror()`, `stderr`, `strerror()`
Synopsis:

N/A

Description:

This linker symbol defines the end of the text segment. This variable isn’t defined in any header file.

Classification:

QNX Neutrino

See also:

brk(), _btext, _edata, _end, sbrk()
 execl()  
Execute a file

Synopsis:

```c
#include <process.h>

int execl( const char * path,
           const char * arg0,
           const char * arg1,
           ...
           const char * argn,
           NULL);
```

Arguments:

- `path` The path of the file to execute.
- `arg0, ..., argn` Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. You must terminate the list with a NULL pointer. The `arg0` argument must point to a filename that’s associated with the process being started.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `execl()` function replaces the current process image with a new process image specified by `path`. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

When a C-language program is executed as a result of this call, it’s entered as a C-language function call as follows:

```c
int main (int argc, char *argv[]);
```
where \textit{argc} is the argument count and \textit{argv} is an array of character pointers to the arguments themselves. In addition, the following variable:

\begin{verbatim}
extern char **environ;
\end{verbatim}

is initialized as a pointer to an array of character pointers to the environment strings. The \textit{argv} and \textit{environ} arrays are each terminated by a null pointer. The null pointer terminating the \textit{argv} array isn’t counted in \textit{argc}.

Multithreaded applications shouldn’t use the \textit{environ} variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the \textit{environ} variable to access that environment variable.

The arguments specified by a program with one of the \textit{exec*} functions are passed on to the new process image in the corresponding \textit{main()} arguments.

The number of bytes available for the new process’s combined argument and environment lists is \texttt{ARG_MAX}.

File descriptors open in the calling process image remain open in the new process image, except for when \texttt{fcntl()}’s \texttt{FD_CLOEXEC} flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by \texttt{close()} while locks not affected by \texttt{close()} aren’t changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to \texttt{SIG_DFL} in the calling process are set to the default action in the new process image. Signals set to \texttt{SIG_IGN} by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren’t preserved and the \texttt{SA_ONSTACK} flag is cleared for all signals.

After a successful call, any functions previously registered by \texttt{atexit()} are no longer registered.
If the path is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of path. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of path. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by setuid()).

Any shared memory segments attached to the calling process image aren’t attached to the new process image.

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see alarm() )
- current working directory
- root directory
- file mode creation mask (see umask() )
- process signal mask (see sigprocmask() )
- pending signal (see sigpending() )
• `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` (see `times()`)

• resource limits

• controlling terminal

• interval timers.

If you call this function from a process with more than one thread, all of the threads are terminated and the new executable image is loaded and executed. No destructor functions are called.

Upon successful completion, the `st_atime` field of the file is marked for update. If the `exec*` function failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. On success, the process image file is considered to be opened with `open()`. The corresponding `close()` is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the `exec*` functions.

**exec*() summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execl()</code></td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, specify the new process environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlp()</code></td>
<td>NULL-terminated argument list, search for the new process in <strong>PATH</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlepe()</code></td>
<td>NULL-terminated argument list, search for the new process in <strong>PATH</strong>, specify the new process environment</td>
<td>No</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>NULL-terminated array of arguments, specify the new process environment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued…*
Function | Description | POSIX?
---|---|---
execvp() | NULL-terminated array of arguments, search for the new process in PATH | Yes
execvpe() | NULL-terminated array of arguments, search for the new process in PATH, specify the new process environment | No

**Returns:**

When `execl()` is successful, it doesn’t return; otherwise, it returns -1 (`errno` is set).

**Errors:**

- E2BIG The argument list and the environment is larger than the system limit of ARG_MAX bytes.
- EACCESS The calling process doesn’t have permission to search a directory listed in `path`, or it doesn’t have permission to execute `path`, or `path`’s filesystem was mounted with the ST_NOEXEC flag.
- ELOOP Too many levels of symbolic links or prefixes.
- ENAMETOOLONG The length of `path` or an element of the `PATH` environment variable exceeds PATH_MAX.
- ENOENT One or more components of the pathname don’t exist, or the `path` argument points to an empty string.
- ENOEXEC The new process image file has the correct access permissions, but isn’t in the proper format.
- ENOMEM There’s insufficient memory available to create the new process.
- ENOTDIR A component of `path` isn’t a directory.
Examples:

Replace the current process with \texttt{myprog} as if a user had typed:
\texttt{myprog ARG1 ARG2}

at the shell:

```
#include <stddef.h>
#include <process.h>

execl( "myprog", "myprog", "ARG1", "ARG2", NULL );
```

In this example, \texttt{myprog} will be found if it exists in the current working directory.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

\texttt{abort()}, \texttt{atexit()}, \texttt{errno}, \texttt{execl()}, \texttt{execlp()}, \texttt{execlpe()}, \texttt{execv()}, \texttt{execve()}, \texttt{execvp()}, \texttt{execvep()}, \texttt{exit()}, \texttt{getenv()}, \texttt{main()}, \texttt{putenv()}, \texttt{spawn()}, \texttt{spawnl()}, \texttt{spawnle()}, \texttt{spawnlp()}, \texttt{spawnlpe()}, \texttt{spawnp()}, \texttt{spawnv()}, \texttt{spawnve()}, \texttt{spawnvp()}, \texttt{spawnvpe()}, \texttt{system()}
execle()

Execute a file

Synopsis:

```c
#include <process.h>

int execle( const char * path,
            const char * arg0,
            const char * arg1,
            ...
            const char * argn,
            NULL,
            const char * envp[] );
```

Arguments:

- **path**
  - The path of the file to execute.

- **arg0, ..., argn**
  - Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. You must terminate the list with a NULL pointer. The arg0 argument must point to a filename that’s associated with the process being started.

- **envp**
  - An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the envp array with a NULL pointer.

Library:

**libc**

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The execle() function replaces the current process image with a new process image specified by path. The new image is constructed from a regular, executable file called the new process image file. No return is
made because the calling process image is replaced by the new process image.

When a C-language program is executed as a result of this call, it’s entered as a C-language function call as follows:

```c
int main (int argc, char *argv[]);
```

where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable:

```c
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The `argv` and `environ` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array isn’t counted in `argc`.

Multithreaded applications shouldn’t use the `environ` variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the `environ` variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding `main()` arguments.

The number of bytes available for the new process’s combined argument and environment lists is `ARG_MAX`.

File descriptors open in the calling process image remain open in the new process image, except for when `fcntl()`’s `FD_CLOEXEC` flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by `close()` while locks not affected by `close()` aren’t changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to `SIG_DFL` in the calling process are set to the default action in the new process image. Signals set to `SIG_IGN` by the calling
process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren’t preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by atexit() are no longer registered.

If the path is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of path. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of path. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by setuid()).

Any shared memory segments attached to the calling process image aren’t attached to the new process image.

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see alarm())
- current working directory
- root directory
- file mode creation mask (see umask())
- process signal mask (see sigprocmask())
- pending signal (see sigpending())
- tms_utime, tms_stime, tms_cutime, and tms_cstime (see times())
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the st_atime field of the file is marked for update. If the exec* function failed but was able to locate the process image file, whether the st_atime field is marked for update is unspecified. On success, the process image file is considered to be opened with open(). The corresponding close() is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the exec* functions.

**exec*() summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td>execl()</td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td>execle()</td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td>execlp()</td>
<td>NULL-terminated argument list, search for the new process in PATH</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
execle()

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td>execlpe()</td>
<td>NULL-terminated argument list, search for the new process in PATH, specify the new process's environment</td>
<td>No</td>
</tr>
<tr>
<td>execv()</td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td>execve()</td>
<td>NULL-terminated array of arguments, specify the new process's environment</td>
<td>Yes</td>
</tr>
<tr>
<td>execvp()</td>
<td>NULL-terminated array of arguments, search for the new process in PATH</td>
<td>Yes</td>
</tr>
<tr>
<td>execvpe()</td>
<td>NULL-terminated array of arguments, search for the new process in PATH, specify the new process's environment</td>
<td>No</td>
</tr>
</tbody>
</table>

Returns:

When `execle()` is successful, it doesn’t return; otherwise, it returns -1 and sets `errno`.

Errors:

- **E2BIG**
  The argument list and the environment is larger than the system limit of ARG_MAX bytes.

- **EACCESS**
  The calling process doesn’t have permission to search a directory listed in `path`, or it doesn’t have permission to execute `path`, or `path`’s filesystem was mounted with the ST_NOEXEC flag.

- **ELOOP**
  Too many levels of symbolic links or prefixes.

- **ENAMETOOLONG**
  The length of `path` or an element of the PATH environment variable exceeds PATH_MAX.

- **ENOENT**
  One or more components of the pathname don’t exist, or the `path` argument points to an empty string.
ENOEXEC The new process’s image file has the correct access permissions, but isn’t in the proper format.

ENOMEM There’s insufficient memory available to create the new process.

ENOTDIR A component of path isn’t a directory.

Examples:

Replace the current process with myprog as if a user had typed:

myprog ARG1 ARG2

at the shell:

#include <unistd.h>
#include <sys/wait.h>

char* env_list[] = { "SOURCE=MYDATA",
                   "TARGET=OUTPUT",
                   "lines=65",
                   NULL
                   };

execle( "myprog",
       "myprog", "ARG1", "ARG2", NULL,
       env_list );

In this example, myprog will be found if it exists in the current working directory. The environment for the invoked program consists of the three environment variables SOURCE, TARGET and lines.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

continued...
### execle()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### See also:

`abort()`, `atexit()`, `errno`, `execl()`, `execlp()`, `execle()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `_exit()`, `exit()`, `getenv()`, `main()`, `putenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`,
**execlp()**

Execute a file

**Synopsis:**

```c
#include <process.h>

int execlp( const char * file,
           const char * arg0,
           const char * arg1,
           ...
           const char * argn,
           NULL );
```

**Arguments:**

- **file**
  
  Used to construct a pathname that identifies the new process image file. If the `file` argument contains a slash character, the `file` argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable `PATH`.

- **arg0, ..., argn**
  
  Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. Terminate the list terminated with a NULL pointer. The `arg0` argument must point to a filename that’s associated with the process.

**Library:**

- **libc**
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `execlp()` function replaces the current process image with a new process image specified by `file`. The new image is constructed from a regular, executable file called the new process image file. No return is
made because the calling process image is replaced by the new process image.

When a C-language program is executed as a result of this call, it’s entered as a C-language function call as follows:

```c
int main (int argc, char *argv[]);
```

where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable:

```c
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The `argv` and `environ` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array isn’t counted in `argc`.

Multithreaded applications shouldn’t use the `environ` variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the `environ` variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding `main()` arguments.

If the process image file isn’t a valid executable object, the contents of the file are passed as standard input to a command interpreter conforming to the `system()` function. In this case, the command interpreter becomes the new process image.

The number of bytes available for the new process’s combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when `ffcntl()`’s FD_CLOEXEC flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by `close()` while locks not affected by `close()` aren’t changed.
Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren’t preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by `atexit()` are no longer registered.

If the `file` is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of `file`. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of `file`. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by `setuid()`).

Any shared memory segments attached to the calling process image aren’t attached to the new process image.

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see alarm())
- current working directory
- root directory
- file mode creation mask (see umask())
- process signal mask (see sigprocmask())
- pending signal (see sigpending())
- tms_utime, tms_stime, tms_cutime, and tms_cstime (see times())
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the st_atime field of the file is marked for update. If the exec* failed but was able to locate the process image file, whether the st_atime field is marked for update is unspecified. On success, the process image file is considered to be opened with open(). The corresponding close() is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the exec* functions.

**exec*() summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td>execl()</td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlp()</code></td>
<td>NULL-terminated argument list, search for the new process in <code>PATH</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, search for the new process in <code>PATH</code>, specify the new process’s environment</td>
<td>No</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>NULL-terminated array of arguments, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvp()</code></td>
<td>NULL-terminated array of arguments, search for the new process in <code>PATH</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvpe()</code></td>
<td>NULL-terminated array of arguments, search for the new process in <code>PATH</code>, specify the new process’s environment</td>
<td>No</td>
</tr>
</tbody>
</table>

### Returns:

When `execlp()` is successful, it doesn’t return; otherwise, it returns -1 and sets `errno`.

### Errors:

- **E2BIG**  
The argument list and the environment is larger than the system limit of `ARG_MAX` bytes.

- **EACCESS**  
The calling process doesn’t have permission to search a directory listed in `file`, or it doesn’t have permission to execute `file`, or `file`’s filesystem was mounted with the `ST_NOEXEC` flag.

- **ELOOP**  
Too many levels of symbolic links or prefixes.
The length of file or an element of the PATH environment variable exceeds PATH_MAX.

ENOMEM There’s insufficient memory available to create the new process.

Enomem: A component of file isn’t a directory.

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abort(), atexit(), errno, execl(), execle(), execlpe(), execv(), execve(), execvp(), execvpe() _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system()
**Synopsis:**

```c
#include <process.h>

int execlpe(const char * file,
             const char * arg0,
             const char * arg1,
             ...;
             const char * argn,
             NULL,
             const char * envp[]);
```

**Arguments:**

- **file**
  
  Used to construct a pathname that identifies the new process image file. If the file argument contains a slash character, the file argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable `PATH`.

- **arg0...argn**
  
  Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. Terminate the list terminated with a NULL pointer. The `arg0` argument must point to a filename that’s associated with the process.

- **envp**
  
  An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the `envp` array with a NULL pointer.

**Library:**

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
execlpe()

Description:

See execl() for further information on the exec*() family of functions.

The execlpe() function replaces the current process image with a new process image specified by file. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

If the new process is a shell script, the first line must start with #!, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The execlpe() function uses the paths listed in the PATH environment variable to locate the program to be loaded, provided that the following conditions are met:

- The argument file identifies the name of program to be loaded.
- If no path character (/) is included in the name, an attempt is made to load the program from one of the paths in the PATH environment variable.
- If PATH isn’t defined, the current working directory is used.
- If a path character (/) is included in the name, the program is loaded from the path specified in file.

The process is started with the arguments specified in the NULL-terminated arguments arg1...argn. arg0 should point to a filename associated with the program being loaded. Only arg0 is required, arg1...argn are optional.

The new process’s environment is specified in envp, a NULL-terminated array of NULL-terminated strings. envp cannot be NULL, but envp[0] can be a NULL pointer if no environment strings are passed.

Each pointer in envp points to a string in the form:
variable=value

that is used to define an environment variable.

The environment is the collection of environment variables whose values have been defined with the `export` shell command, the `env` utility, or by the successful execution of the `putenv()` or `setenv()` functions.

A program may read these values with the `getenv()` function.

An error is detected when the program cannot be found.

If the `file` is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process is set to the owner ID of `file`. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of `file`. The real user ID, real group ID and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process are saved as the saved set-user ID and the saved set-group ID used by `setuid()`.

**exec*() summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execl()</code></td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlp()</code></td>
<td>NULL-terminated argument list, search for the new process in PATH</td>
<td>Yes</td>
</tr>
<tr>
<td><code>exclpe()</code></td>
<td>NULL-terminated argument list, search for the new process in PATH, specify the new process’s environment</td>
<td>No</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued…*
**execlepe()**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execve()</code></td>
<td>NULL-terminated array of arguments, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvp()</code></td>
<td>NULL-terminated array of arguments, search for the new process in <strong>PATH</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvpe()</code></td>
<td>NULL-terminated array of arguments, search for the new process in <strong>PATH</strong>, specify the new process’s environment</td>
<td>No</td>
</tr>
</tbody>
</table>

**Returns:**

When `execlepe()` is successful, it doesn’t return; otherwise, it returns -1 and sets `errno`.

**Errors:**

- **E2BIG** The argument list and the environment is larger than the system limit of ARG_MAX bytes.
- **EACCESS** The calling process doesn’t have permission to search a directory listed in `file`, or it doesn’t have permission to execute `file`, or `file`’s filesystem was mounted with the ST_NOEXEC flag.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG** The length of `file` or an element of the **PATH** environment variable exceeds PATH_MAX.
- **ENOENT** One or more components of the pathname don’t exist, or the `file` argument points to an empty string.
- **ENOMEM** There’s insufficient memory available to create the new process.
- **ENOTDIR** A component of `file` isn’t a directory.
execlepe() © 2007, QNX Software Systems GmbH & Co. KG.

Classification:
QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
abort(), atexit(), errno, execl(), execle(), execlp(), execv(), execve(), execvp(), execvpe(), _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnype(), system()
**Synopsis:**

```c
#include <process.h>

int execv( const char * path,
            char * const argv[] );
```

**Arguments:**

- `path`: A path name that identifies the new process image file.
- `argv`: An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in `argv[0]` must point to a filename that’s associated with the process being started.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `execv()` function replaces the current process image with a new process image specified by `path`. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

When a C-language program is executed as a result of this call, it’s entered as a C-language function call as follows:

```c
int main (int argc, char *argv[]);
```

where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable:
extern char **environ;

is initialized as a pointer to an array of character pointers to the environment strings. The `argv` and `environ` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array isn’t counted in `argc`.

Multithreaded applications shouldn’t use the `environ` variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the `environ` variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding `main()` arguments.

The environment for the new process image is taken from the external variable `environ` in the calling process.

The number of bytes available for the new process’s combined argument and environment lists is `ARG_MAX`.

File descriptors open in the calling process image remain open in the new process image, except for when `fcntl()`’s `FD_CLOEXEC` flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by `close()` while locks not affected by `close()` aren’t changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren’t preserved and the `SA_ONSTACK` flag is cleared for all signals.

After a successful call, any functions previously registered by `atexit()` are no longer registered.
If the *path* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *path*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by *setuid()*).

Any shared memory segments attached to the calling process image aren’t attached to the new process image.

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see *alarm()*)
- current working directory
- root directory
- file mode creation mask (see *umask()*)
- process signal mask (see *sigprocmask()*)
- pending signal (see *sigpending()*).
execv() summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td>execcl()</td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td>execle()</td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td>execclp()</td>
<td>NULL-terminated argument list, search for the new process in PATH</td>
<td>Yes</td>
</tr>
<tr>
<td>execlpe()</td>
<td>NULL-terminated argument list, search for the new process in PATH, specify the new process’s environment</td>
<td>No</td>
</tr>
<tr>
<td>execv()</td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td>execve()</td>
<td>NULL-terminated array of arguments, specify the new process’s environment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
Function | Description | POSIX?
--- | --- | ---
execvp() | NULL-terminated array of arguments, search for the new process in PATH | Yes
execvpe() | NULL-terminated array of arguments, search for the new process in PATH, specify the new process’s environment | No

Returns:

When `execv()` is successful, it doesn’t return; otherwise, it returns -1 and sets `errno`.

Errors:

- **E2BIG**: The argument list and the environment is larger than the system limit of ARG_MAX bytes.
- **EACCESS**: The calling process doesn’t have permission to search a directory listed in `path`, or it doesn’t have permission to execute `path`, or `path`’s filesystem was mounted with the ST_NOEXEC flag.
- **ELOOP**: Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG**: The length of `path` or an element of the PATH environment variable exceeds PATH_MAX.
- **ENOENT**: One or more components of the pathname don’t exist, or the `path` argument points to an empty string.
- **ENOEXEC**: The new process’s image file has the correct access permissions, but isn’t in the proper format.
- **ENOMEM**: There’s insufficient memory available to create the new process.
- **ENOTDIR**: A component of `path` isn’t a directory.
execv()

Examples:

```c
#include <stddef.h>
#include <process.h>

char* arg_list[] = { "myprog", "ARG1", "ARG2", NULL };
execv( "myprog", arg_list );
```

The preceding invokes `myprog` as if the user entered:

```
myprog ARG1 ARG2
```

as a command at the shell. The program will be found if `myprog`
exists in the current working directory.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
</tr>
<tr>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`abort()`, `atexit()`, `errno`, `execl()`, `execle()`, `exclpe()`, `execve()`,
`execvp()`, `execvpe()`, `execve()`, `execlp()`, `execvp()`, `spawn()`,
`spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`,
`spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`, `system()`
Synopsis:

```c
#include <process.h>

int execve( const char * path,
            char * const argv[],
            char * const envp[] );
```

Arguments:

- `path`: A path name that identifies the new process image file.
- `argv`: An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in `argv[0]` must point to a filename that’s associated with the process being started.
- `envp`: An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the `envp` array with a NULL pointer.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `execve()` function replaces the current process image with a new process image specified by `path`. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

When a C-language program is executed as a result of this call, it’s entered as a C-language function call as follows:

```c
int main (int argc, char *argv[]);
```
where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable:

```c
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The `argv` and `environ` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array isn’t counted in `argc`.

Multithreaded applications shouldn’t use the `environ` variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the `environ` variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding `main()` arguments.

The number of bytes available for the new process’s combined argument and environment lists is `ARG_MAX`.

File descriptors open in the calling process image remain open in the new process image, except for when `fcntl()`’s `FD_CLOEXEC` flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by `close()` while locks not affected by `close()` aren’t changed.

Directory streams open in the calling process image are closed in the new process image.

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren’t preserved and the `SA_ONSTACK` flag is cleared for all signals.

After a successful call, any functions previously registered by `atexit()` are no longer registered.
If the *path* is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of *path*. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of *path*. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by `setuid()`).

Any shared memory segments attached to the calling process image aren’t attached to the new process image.

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs
- time left until an alarm clock signal (see *alarm()*
- current working directory
- root directory
- file mode creation mask (see *umask()*
- process signal mask (see *sigprocmask()*
- pending signal (see *sigpending()*

September 10, 2007

QNX Neutrino Functions and Macros 609
execve()

- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` (see `times()`)
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the `st_atime` field of the file is marked for update. If the `exec*` failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. On success, the process image file is considered to be opened with `open()`. The corresponding `close()` is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the `exec*` functions.

exec*() summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execl()</code></td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlp()</code></td>
<td>NULL-terminated argument list, search for the new process in <code>PATH</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlepe()</code></td>
<td>NULL-terminated argument list, search for the new process in <code>PATH</code>, specify the new process’s environment</td>
<td>No</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>NULL-terminated array of arguments, specify the new process’s environment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td>execvp()</td>
<td>NULL-terminated array of arguments, search for the new process in <strong>PATH</strong></td>
<td>Yes</td>
</tr>
<tr>
<td>execvpe()</td>
<td>NULL-terminated array of arguments, search for the new process in <strong>PATH</strong>, specify the new process’s environment</td>
<td>No</td>
</tr>
</tbody>
</table>

#### Returns:

When `execve()` is successful, it doesn’t return; otherwise, it returns -1 and sets `errno`.

#### Errors:

- **E2BIG** The argument list and the environment is larger than the system limit of ARG_MAX bytes.
- **EACCESS** The calling process doesn’t have permission to search a directory listed in `path`, or it doesn’t have permission to execute `path`, or `path`’s filesystem was mounted with the ST_NOEXEC flag.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG** The length of `path` or an element of the **PATH** environment variable exceeds PATH_MAX.
- **ENOENT** One or more components of the pathname don’t exist, or the `path` argument points to an empty string.
- **ENOEXEC** The new process’s image file has the correct access permissions, but isn’t in the proper format.
- **ENOMEM** There’s insufficient memory available to create the new process.
- **ENOTDIR** A component of `path` isn’t a directory.
execve()

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abort(), atexit(), errno, execl(), execle(), execlp(), execle(), execvp(), execvpe(), _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system()
Synopsis:

```
#include <process.h>

int execvp( const char * file,
            char * const argv[] );
```

Arguments:

- **file**: Used to construct a pathname that identifies the new process image file. If the `file` argument contains a slash character, the `file` argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable `PATH`.

- **argv**: An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in `argv[0]` must point to a filename that’s associated with the process being started.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `execvp()` function replaces the current process image with a new process image specified by `file`. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

When a C-language program is executed as a result of this call, it’s entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```
where `argc` is the argument count and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable:

```c
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The `argv` and `environ` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array isn’t counted in `argc`.

Multithreaded applications shouldn’t use the `environ` variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable is considered a use of the `environ` variable to access that environment variable.

The arguments specified by a program with one of the exec functions are passed on to the new process image in the corresponding `main()` arguments.

If the process image file isn’t a valid executable object, the contents of the file are passed as standard input to a command interpreter conforming to the `system()` function. In this case, the command interpreter becomes the new process image.

The environment for the new process image is taken from the external variable `environ` in the calling process.

The number of bytes available for the new process’s combined argument and environment lists is `ARG_MAX`.

File descriptors open in the calling process image remain open in the new process image, except for when `fcntl()`’s `FD_CLOEXEC` flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by `close()` while locks not affected by `close()` aren’t changed.

Directory streams open in the calling process image are closed in the new process image.
Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image. After a successful call, alternate signal stacks aren’t preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by `atexit()` are no longer registered.

If the `file` is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of `file`. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of `file`. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process image are saved (as the saved set-user ID and the saved set-group ID used by `setuid()`).

Any shared memory segments attached to the calling process image aren’t attached to the new process image.

The new process also inherits at least the following attributes from the calling process image:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- supplementary group IDs


- time left until an alarm clock signal (see `alarm()`)
- current working directory
- root directory
- file mode creation mask (see `umask()`)
- process mode creation mask (see `sigprocmask()`)
- pending signal (see `sigpending()`)
- `tms_utime, tms_stime, tms_cutime, and tms_cstime` (see `times()`)
- resource limits
- controlling terminal
- interval timers.

A call to this function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions are called.

Upon successful completion, the `st_atime` field of the file is marked for update. If the `exec` function failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. On success, the process image file is considered to be opened with `open()`. The corresponding `close()` is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the `exec*` functions.

**`exec*()` summary**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execl()</code></td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, specify the new process's environment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

`continued…`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execvp()</code></td>
<td>NULL-terminated argument list, search for the new process in PATH</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvpe()</code></td>
<td>NULL-terminated argument list, search for the new process in PATH, specify the new process’s environment</td>
<td>No</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>NULL-terminated array of arguments, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvp()</code></td>
<td>NULL-terminated array of arguments, search for the new process in PATH</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvpe()</code></td>
<td>NULL-terminated array of arguments, search for the new process in PATH, specify the new process’s environment</td>
<td>No</td>
</tr>
</tbody>
</table>

**Returns:**

When `execvp()` is successful, it doesn’t return; otherwise, it returns -1 and sets `errno`.

**Errors:**

- **E2BIG** The argument list and the environment is larger than the system limit of `ARG_MAX` bytes.
- **EACCESS** The calling process doesn’t have permission to search a directory listed in `file`, or it doesn’t have permission to execute `file`, or `file`’s filesystem was mounted with the ST_NOEXEC flag.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG** The length of `file` or an element of the `PATH` environment variable exceeds `PATH_MAX`. 
ENOENT 	One or more components of the pathname don’t exist, or the \textit{file} argument points to an empty string.

ENOMEM 	There’s insufficient memory available to create the new process.

ENOTDIR 	A component of \textit{file} isn’t a directory.

\textbf{Classification:}

POSIX 1003.1

\begin{center}
\textbf{Safety}
\begin{tabular}{ll}
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\end{tabular}
\end{center}

\textbf{See also:}

\texttt{abort()}, \texttt{atexit()}, \texttt{errno}, \texttt{exce\textunderscore l()}, \texttt{exec\textunderscore cle()}, \texttt{exec\textunderscore clp()}, \texttt{exec\textunderscore lpe()}, \texttt{exec\textunderscore v()}, \texttt{exec\textunderscore ve()}, \texttt{exec\textunderscore vpe()}, \texttt{\_exit()}, \texttt{exit()}, \texttt{getenv()}, \texttt{main()}, \texttt{putenv()}, \texttt{spawn()}, \texttt{spawn\textunderscore l()}, \texttt{spawn\textunderscore le()}, \texttt{spawn\textunderscore lp()}, \texttt{spawn\textunderscore lpe()}, \texttt{spawnp()}, \texttt{spawnv()}, \texttt{spawn\textunderscore v()}, \texttt{spawn\textunderscore vpe()}, \texttt{system()}
Synopsis:

```c
#include <process.h>

int execvpe( const char * file,
             char * const argv[],
             char * const envp[] );
```

Arguments:

- **file**: Used to construct a pathname that identifies the new process image file. If the `file` argument contains a slash character, the `file` argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable `PATH`.

- **argv**: An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in `argv[0]` must point to a filename that’s associated with the process being started.

- **envp**: An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the `envp` array with a NULL pointer.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

See `execl()` for further information on the `exec*()` family of functions.

The `execvpe()` function replaces the current process image with a new process image specified by `file`. The new image is constructed from a
regular, executable file called the new process image file. No return is
made because the calling process image is replaced by the new
process image.

If the new process is a shell script, the first line must start with #!,
followed by the path and arguments of the shell to be run to interpret
the script. The script must also be marked as executable.

The `execvpe()` function uses the paths listed in the `PATH` environment
variable to locate the program to be loaded, provided that the
following conditions are met:

- The `file` argument identifies the name of program to be loaded.
- If no path character (/) is included in the name, an attempt is made
to load the program from one of the paths in the `PATH`
environment variable.
- If `PATH` isn’t defined, the current working directory is used.
- If a path character (/) is included in the name, the program is
  loaded from the path specified in `file`.

The process is started with the argument specified in `argv`, a
NULL-terminated array of NULL-terminated strings. The `argv[0]`
entry should point to a filename associated with the program being
loaded. The `argv` argument can’t be NULL but `argv[0]` can be NULL if
no arguments are required.

The new process’s environment is specified in `envp`, a
NULL-terminated array of NULL-terminated strings. `envp` cannot be
NULL, but `envp[0]` can be a NULL pointer if no environment strings
are passed.

Each pointer in `envp` points to a string in the form:

```
variable=value
```

that is used to define an environment variable.

The environment is the collection of environment variables whose
values have been defined with the `export` shell command, the `env`
utility, or by the successful execution of the `putenv()` or `setenv()` functions.

A program may read these values with the `getenv()` function.

An error is detected when the program cannot be found.

If the file is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process is set to the owner ID of file. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of file. The real user ID, real group ID and supplementary group IDs of the new process remain the same as those of the calling process. The effective user ID and effective group ID of the new process are saved as the saved set-user ID and the saved set-group ID used by `setuid()`.

### exec*( ) summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execl()</code></td>
<td>NULL-terminated argument list</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execle()</code></td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlp()</code></td>
<td>NULL-terminated argument list, search for the new process in PATH</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlpe()</code></td>
<td>NULL-terminated argument list, search for the new process in PATH, specify the new process’s environment</td>
<td>No</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>NULL-terminated array of arguments</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>NULL-terminated array of arguments, specify the new process’s environment</td>
<td>Yes</td>
</tr>
<tr>
<td><code>execvp()</code></td>
<td>NULL-terminated array of arguments, search for the new process in PATH</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued…*
**execvpe()**

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<tr>
<td>execvpe()</td>
<td>NULL-terminated array of arguments, search for the new process in PATH, specify the new process’s environment</td>
<td>No</td>
</tr>
</tbody>
</table>

**Returns:**

When execvpe() is successful, it doesn’t return; otherwise, it returns -1 and sets errno.

**Errors:**

- **E2BIG** The argument list and the environment is larger than the system limit of ARG_MAX bytes.
- **EACCESS** The calling process doesn’t have permission to search a directory listed in file, or it doesn’t have permission to execute file, or file’s filesystem was mounted with the ST_NOEXEC flag.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **ENAME_TOO_LONG** The length of file or an element of the PATH environment variable exceeds PATH_MAX.
- **ENOENT** One or more components of the pathname don’t exist, or the file argument points to an empty string.
- **ENOMEM** There’s insufficient memory available to create the new process.
- **ENOTDIR** A component of file isn’t a directory.
execvpe()

Classification:
QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
abort(), atexit(), errno, execl(), execlp(), execvp(), execve(), execvp(), _exit(), exit(), getenv(), main(), putenv(), spawnl(), spawnlp(), spawnle(), spawnlppe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system()
**exit()**

Terminate the program

**Synopsis:**

```c
#include <stdlib.h>

void _exit( int status );
```

**Arguments:**

- **status**  
  The exit status to use for the program. The value may be zero, EXIT_STATUS, EXIT_FAILURE or any other value. Note that only the least significant bits (i.e. status and 0377) may be available to a waiting parent process.

**Library:**

- **libc**  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `_exit()` function causes normal program termination to occur.

The functions registered with `atexit()` aren’t called when you use `_exit()` to terminate a program. If you want those functions to be called, use `exit()` instead.

The `_exit()` function does the following when a process terminates for any reason:

1. Closes all open file descriptors and directory streams in the calling process.

2. Notifies the parent process of the calling process if the parent called `wait()` or `waitpid()`. The low-order 8 bits of `status` are made available to the parent via `wait()` or `waitpid()`.

3. Saves the exit `status` if the parent process of the calling process isn’t executing a `wait()` or `waitpid()` function. If the parent calls `wait()` or `waitpid()` later, this status is returned immediately.
__exit__

4 Sends a SIGHUP signal to the calling process’s children; this can indirectly cause the children to exit if they don’t handle SIGHUP. Children of a terminated process are assigned a new parent process.

5 Sends a SIGCHLD signal to the parent process.

6 Sends a SIGHUP signal to each process in the foreground process group if the calling process is the controlling process for the controlling terminal of that process group.

7 Disassociates the controlling terminal from the calling process’s session if the process is a controlling process, allowing it to be acquired by a new controlling process.

8 If the process exiting causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly-orphaned process group.

Returns:

The __exit__() function doesn’t return.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    FILE *fp;

    if( argc <= 1 ) {
        fprintf( stderr, "Missing argument\n" );
        exit( EXIT_FAILURE );
    }

    fp = fopen( argv[1], "r" );
    if( fp == NULL ) {
        fprintf( stderr, "Unable to open '%s'\n", argv[1] );
        _exit( EXIT_FAILURE );
    }

    fclose( fp );
```

At this point, calling _exit() is the same as calling
return EXIT_SUCCESS;...
*/
_exit( EXIT_SUCCESS );
}

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abort(), atexit(), close(), exec(), execv(), execvp(), execve(), execvp(), execvpe(), _exit(), getenv(), main(), putenv(),
sigaction(), signal(), spawn(), spawnl(), spawnle(), spawnlp(),
spawnlpe(), spawnn(), spawnv(), spawnve(), spawnvp(), spawnvpe(),
system(), wait(), waitpid()
Synopsis:

```c
#include <stdlib.h>

void exit( int status );
```

Arguments:

- `status` The exit status to use for the program. The value may be zero, EXIT_STATUS, EXIT_FAILURE or any other value. Note that only the least significant bits (i.e. `status` and 0377) may be available to a waiting parent process.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `exit()` function causes the calling program to exit normally. When a program exits normally:

1. All functions registered with the `atexit()` function are called.
2. All open file streams (those opened by `fopen()`, `fdopen()`, `freopen()`, or `popen()`) are flushed and closed.
3. All temporary files created by the `tmpfile()` function are removed.
4. The return `status` is made available to the parent process; `status` is typically set to EXIT_SUCCESS to indicate successful termination and set to EXIT_FAILURE or some other value to indicate an error.
Returns:

The `exit()` function doesn’t return.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    FILE *fp;

    if( argc <= 1 ) {
        fprintf( stderr, "Missing argument\n" );
        exit( EXIT_FAILURE );
    }

    fp = fopen( argv[1], "r" );
    if( fp == NULL ) {
        fprintf( stderr, "Unable to open '%s'\n", argv[1] );
        exit( EXIT_FAILURE );
    }
    fclose( fp );
    exit( EXIT_SUCCESS );

    /*
     * You’ll never get here; this prevents compiler
     * warnings about "function has no return value".
     */
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

abort(), atexit(), _exit(), main()
exp(), expf()

Compute the exponential function of a number

Synopsis:

```c
#include <math.h>

double exp( double x );

float expf( float x );
```

Arguments:

- `x` The number for which you want to calculate the exponential.

Library:

```c
libm
```

Use the `-l m` option to `qcc` to link against this library.

Description:

The `exp()` function computes the exponential function of `x` ($e^x$).

A range error occurs if the magnitude of `x` is too large.

Returns:

The exponential value of `x`.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

---

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", exp(.5) );
}
```
return EXIT_SUCCESS;
}

produces the output:

1.648721

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The value of \( \expm1(x) \) may be more accurate than \( \exp(x) - 1.0 \) for small values of \( x \).

See also:

\( \texttt{errno, expm1, log()} \)
**expm1(), expm1f()**

Compute the exponential of a number, then subtract 1

**Synopsis:**

```c
#include <math.h>

double expm1 ( double x );

float expm1f ( float x );
```

**Arguments:**

- `x` The number for which you want to calculate the exponential minus one.

**Library:**

- `libm`
  
  Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `expm1()` and `expm1f()` functions compute the exponential of `x`, minus 1 (`e^x - 1`).

A range error occurs if the magnitude of `x` is too large.

The value of `expm1(x)` may be more accurate than `exp(x) - 1.0` for small values of `x`.

The `expm1()` and `log1p()` functions are useful for financial calculations of `(((1+x) ** n) - 1) / x`, namely:

```
expm1(n * log1p(x)) / x
```

when `x` is very small (for example, when performing calculations with a small daily interest rate). These functions also simplify writing accurate inverse hyperbolic functions.

**Returns:**

The exponential value of `x`, minus 1.
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;
    a = 2;
    b = expm1(a);
    printf("(e \^ \%f) -1 is \%f \n", a, b);
    return(0);
}
```

produces the output:

```
(e \^ 2.000000) -1 is 6.389056
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
**See also:**

`exp()`, `log1p()`
C Library — F to H
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td><code>abort()</code> to <code>expm1f()</code></td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td><code>fabs()</code> to <code>hypotf()</code></td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to <code>lround()</code></td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td><code>main()</code> to <code>outle32()</code></td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td><code>pathconf()</code> to <code>ruserok()</code></td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td><code>sbrk()</code> to <code>system()</code></td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td><code>tan()</code> to <code>ynf()</code></td>
</tr>
</tbody>
</table>
**fabs(), fabsf()**  
© 2007, QNX Software Systems GmbH & Co. KG.

Compute the absolute value of a double number

**Synopsis:**
```c
#include <math.h>

double fabs( double x );
float fabsf( float x );
```

**Arguments:**

- `x` The number you want the absolute value of.

**Library:**

- `libm`

  Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the absolute value of `x`.

**Returns:**

The absolute value of `x`.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**
```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%.1f %.1f\n", fabs(.5), fabs(-.5) );
    return EXIT_SUCCESS;
}
```

638 C Library — F to H September 10, 2007
produces the output:

0.500000 0.500000

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abs(), cabs(), labs()
fcfgopen()

Open a configuration file

Synopsis:

```c
#include <cfgopen.h>

FILE * fcfgopen( const char * path,
    const char * mode,
    int location,
    const char * historical,
    char * namebuf,
    int nblen );
```

Arguments:

- **path**: The name of the configuration file that you want to open.
- **mode**: A string that describes the mode to open in; see `fopen()`.
- **location**: Flags that describe how the path is constructed. See “Search condition flags” in the documentation for `cfgopen()`.
- **historical**: A optional file to open as a last resort if none of the criteria for finding the path is met. This string works like a path search order, and lets you search more than one location. You can also specify %H to substitute the hostname value into the string. Specify NULL to ignore this option.
- **namebuf**: A buffer to save the pathname in. Specify NULL to ignore this option.
- **nblen**: The length of the buffer pointed to by `namebuf`. Specify 0 to ignore this option.

Library:

- **libc**

Use the -l c option to gcc to link against this library. This library is usually included automatically.
This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:
The `fcfgopen()` function is similar to `cfgopen()` with these exceptions:

- The CFGFILE_NOFD flag isn’t valid.
- The values for `flags` described in `open()` aren’t valid.

Returns:
A valid `fd` if CFGFILE_NOFD isn’t specified, a nonnegative value if CFGFILE_NOFD is specified, or -1 if an error occurs.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
`cfgopen()`, `confstr()`,
`

mib.txt, snmpd.conf` in the `Utilities Reference`
fchmod()

Change the permissions for a file

Synopsis:

```c
#include <sys/types.h>
#include <sys/stat.h>

int fchmod( int fd,
           mode_t mode );
```

Arguments:

- `fd` A file descriptor for the file whose permissions you want to change.
- `mode` The new permissions for the file. For more information, see “Access permissions” in the documentation for `stat()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fchmod()` function changes the permissions for a file referred to by `fd` to be the settings in the mode given by `mode`.

If the effective user ID of the calling process is equal to the file owner, or the calling process has appropriate privileges (for example, the superuser), `fchmod()` sets the S_ISUID, S_ISGID and the file permission bits, defined in the `<sys/stat.h>` header file, from the corresponding bits in the `mode` argument. These bits define access permissions for the user associated with the file, the group associated with the file, and all others.

For a regular file, if the calling process doesn’t have appropriate privileges, and if the group ID of the file doesn’t match the effective group ID, the S_ISGID (set-group-ID on execution) bit in the file’s mode is cleared upon successful return from `fchmod()`.
Changing the permissions has no any effect on any file descriptors for files that are already open.

If `fchmod()` succeeds, the `st_ctime` field of the file is marked for update.

**Returns:**

0    Success.

-1   An error occurred (`errno` is set).

**Errors:**

EBADF    Invalid file descriptor.
ENOSYS    The `fchmod()` function isn’t implemented for the filesystem specified by `fd`.
EPERM    The effective user ID doesn’t match the owner of the file, and the calling process doesn’t have appropriate privileges.
EROFS    The referenced file resides on a read-only filesystem.

**Examples:**

```c
/*
 * Change the permissions of a list of files
 * to be read/write by the owner only
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;
    int fd;

    for( i = 1; i < argc; i++ ) {
        if( ( fd = open( argv[i], O_RDONLY ) ) == -1 ) {
            perror( argv[i] );
        }
```
fchmod()

```c
    ecode++; 
    } else if( fchmod( fd, S_IRUSR | S_IWUSR ) == -1 ) { 
        perror( argv[i] );
        ecode++; 
    } 
    close( fd );
    return ecode;
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`chmod()`, `chown()`, `errno`, `fchown()`, `fstat()`, `open()`, `stat()`
fchown()
Change the user ID and group ID of a file

Synopsis:
#include <sys/types.h>
#include <unistd.h>

int fchown( int fd,
            uid_t owner,
            gid_t group );

Arguments:

fd    A file descriptor for the file whose ownership you want to change.
owner  The user ID of the new owner.
group  The group ID of the new owner.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The fchown() function changes the user ID and group ID of the file referenced by fd to be the numeric values contained in owner and group, respectively.

Only processes with an effective user ID equal to the user ID of the file, or with appropriate privileges (for example, the superuser) may change the ownership of a file.

The _POSIX_CHOWN_RESTRICTED flag is enforced. This means that only the superuser may change the ownership of a file. The group of a file may be changed by the superuser, or also by a process with the effective user ID equal to the user ID of the file, if (and only if) owner is equal to the user ID of the file and group is equal to the effective group ID of the calling process.
If the `fd` argument refers to a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode are cleared if the function is successful.

If `fchown()` succeeds, the `st_ctime` field of the file is marked for update.

**Returns:**

- `0` Success.
- `-1` An error occurred (`errno` is set).

**Errors:**

- **EBADF** Invalid file descriptor.
- **EPERM** The effective user ID doesn’t match the owner of the file, or the calling process doesn’t have appropriate privileges.
- **EROFS** The named file resides on a read-only filesystem.

**Examples:**

```c
/*
 * Change the ownership of a list of files
 * to the current user/group
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;
    int fd;

    for( i = 1; i < argc; i++ ) {
        if( ( fd = open( argv[i], O_RDONLY ) ) == -1 ) {
            perror( argv[i] );
            ecode++;
        } else if( fchown( fd, getuid(), getgid() ) == -1 ) {
            perror( argv[i] );
    }
}
```c
fchown()

ecode++;
}
close( fd );
return ecode;
}

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`chmod()`, `chown()`, `errno`, `fchmod()`, `fstat()`, `fchown()`, `open()`, `stat()`
fclose()
Close a stream

Synopsis:

```c
#include <stdio.h>

int fclose( FILE* fp );
```

Arguments:

- `fp` The stream you want to close.

Library:

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fclose()` function closes the stream specified by `fp`. Any unwritten, buffered data is flushed before the file is closed. Any unread, buffered data is discarded.

If the associated buffer was automatically allocated, it’s deallocated.

Returns:

0 for success, or `EOF` if an error occurred (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    fp = fopen( "stdio.h", "r" );
    if( fp != NULL ) {
        fclose( fp );
        return EXIT_SUCCESS;
    }
}
fclose()
fcloseall()

Close all open stream files

Synopsis:

```c
#include <stdio.h>

int fcloseall( void );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fcloseall()` function closes all open streams, except `stdin`, `stdout` and `stderr`. This includes streams created (and not yet closed) by `fdopen()`, `fopen()` and `freopen()`.

Returns:

0

Errors:

If an error occurs, `errno` is set.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "The number of files closed is %d\n", fcloseall() );
    return EXIT_SUCCESS;
}
```

Classification:

QNX 4
fcloseall()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fclose(), fdopen(), fopen(), freopen()
**Synopsis:**

```c
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl( int fildes,
           int cmd,
           ... );
```

**Arguments:**

- `fildes`  The descriptor for the file you want to control.
- `cmd`  The command to execute; see below.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `fcntl()` function provides control over the open file referenced by file descriptor `fildes`. To establish a lock with this function, open with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

The type of control is specified by the `cmd` argument, which may require a third data argument (`arg`). The `cmd` argument is defined in `<fcntl.h>`, and includes at least the following values:

- `FALLOCSP`  If the size of the file is less than the number of bytes specified by the extra `arg` argument, extend the file with NUL characters.
- `FDUPFD`  Allocate and return a new file descriptor that’s the lowest numbered available (i.e. not already open) file descriptor greater than or equal to the third
argument, arg, taken as an \texttt{int}. The new file descriptor refers to the same file as \texttt{fildes}, and shares any locks.

**F\_FREESP**  
Truncate the file to the size (in bytes) specified by the extra argument, \texttt{arg}.

**F\_GETFD**  
Get the file descriptor flags associated with the file descriptor \texttt{fildes}. File descriptor flags are associated with a single file descriptor, and don’t affect other file descriptors referring to the same file.

**F\_GETFL**  
Get the file status flags and the file access modes associated with \texttt{fildes}. The flags and modes are defined in \texttt{<fcntl.h>}. The file status flags (see \texttt{open()} for more detailed information) are:

- \texttt{O\_APPEND} — Set append mode.
- \texttt{O\_NONBLOCK} — No delay.

The file access modes are:

- \texttt{O\_RDONLY} — Open for reading only.
- \texttt{O\_RDWR} — Open for reading and writing.
- \texttt{O\_WRONLY} — Open for writing only.

**F\_GETLK**  
Get the first lock that blocks the lock description pointed to by the third argument, arg, taken as a pointer to type \texttt{struct flock} (defined in \texttt{<fcntl.h>}). For more information, see the \texttt{flock} structure section below. The information returned overwrites the information passed to \texttt{fcntl()} in the structure pointed to by \texttt{arg}.

If no lock is found that prevents this lock from being created, the structure is left unchanged, except for the lock type, which is set to \texttt{F\_UNLCK}. If a lock is found, the \texttt{l\_pid} member of the structure
pointed to by arg is set to the process ID of the
process holding the blocking lock and whence is set to SEEK_SET.

F_SETFD Set the file descriptor flags associated with fildes to
the third argument, arg, taken as type int. See the
above discussion for more details.

The only defined file descriptor flag is:

FD_CLOEXEC When this flag is clear, the file
remains open across spawn*() or
exec*() calls; else the file is
closed.

F_SETFL Set the file status flags, as shown above, for the
open file description associated with fildes from the
corresponding bits in the third argument, arg, taken
as type int. You can’t use this function to change
the file access mode. All bits set in arg, other than
the file status bits, are ignored.

F_SETLK Set or clear a file segment lock, according to the
lock description pointed to by the third argument,
arg, taken as a pointer to type struct flock, as
defined in the header file <fcntl.h>, and
documented below. This command is used to create
the following locks (defined in <fcntl.h>):

F_RDLCK Shared or read locks.
F_UNLCK Remove either type of lock.
F_WRLCK Exclusive or write locks.

If a lock can’t be set, fcntl() returns immediately.

F_SETLKW This command is the same as F_SETLK, except that
when a lock is blocked by other locks, the process
waits until the request can be satisfied. If a signal
that’s to be caught is received while fcntl() is
waiting for a region, the call is interrupted without performing the lock operation, and `fcntl()` returns -1 with `errno` set to EINTR.

**flock structure**

The `flock` structure contains at least the following members:

- `short l_type` One of `F_RDLCK`, `F_WRLCK` or `F_UNLCK`.
- `short l_whence` One of the following flags that specify where the relative offset, `l_start`, is measured from:
  - SEEK_CUR: Current seek position.
  - SEEK_END: End of file.
  - SEEK_SET: Start of file.
- `off_t l_start` Relative offset in bytes.
- `off_t l_len` Consecutive bytes to lock; if 0, then until EOF; if negative, the preceding bytes up to, but not including, the start byte.
- `pid_t l_pid` Process ID of the process holding the lock, returned when `cmd` is `F_GETLK`.

When a shared lock is set on a segment of a file, other processes can set shared locks on the same segment, or a portion of it. A shared lock prevents other processes from setting exclusive locks on any portion of the protected area. A request for a shared lock fails if the file was opened write-only.

An exclusive lock prevents any other process from setting a shared or an exclusive lock on a portion of the protected area. A request for an exclusive lock fails if the file was opened read-only.

Locks may start and extend beyond the current end of file, but may not start or extend before the beginning of the file; to attempt to do so...
is an error. A lock extends to “infinity” (the largest possible value for
the file offset) if _len _is set to zero. If _whence _and _start _point to
the beginning of the file, and _len _is zero, the entire file is locked.

The calling process may have only one type of lock set for each byte
of a file. Before successfully returning from an F_SETLK or
F_SETLKW request, the previous lock type (if any) for each byte in
the specified lock region is replaced by the new lock type. All locks
associated with a file for a given process are removed when a file
descriptor for that file is closed by the process, or the process holding
the file descriptor terminates. Locks aren’t inherited by a child
process using the _fork() _function. However, locks are inherited across
_execl*() _or _spawn*() _calls.

---

A potential for deadlock occurs if a process controlling a locked
region is put to sleep by attempting to lock another process’s locked
region. If the system detects that sleeping until a locked region is
unlocked would cause a deadlock, _fcntl() _fails with EDEADLK.
However, the system can’t always detect deadlocks in the network
case, and care should be exercised in the design of your application
for this possibility.
Locking is a protocol designed for updating a file shared among concurrently running applications. Locks are only advisory, that is, they don’t prevent an errant or poorly-designed application from overwriting a locked region of a shared file. An application should use locks to indicate regions of a file that are to be updated by the application, and it should respect the locks of other applications.

The following functions ignore locks:

- `chsize`
- `ltrunc`
- `open`
- `read`
- `sopen`
- `write`

**Returns:**

-1 if an error occurred (errno is set). The successful return value(s) depend on the request type specified by arg, as shown in the following table:

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_DUPFD</td>
<td>A new file descriptor.</td>
</tr>
<tr>
<td>F_GETFD</td>
<td>Value of the file descriptor flags (never a negative value).</td>
</tr>
<tr>
<td>F_GETFL</td>
<td>Value of the file status flags and access modes as shown above (never a negative value).</td>
</tr>
<tr>
<td>F_GETLK</td>
<td>Value other than -1.</td>
</tr>
<tr>
<td>F_SETFD</td>
<td>Value other than -1.</td>
</tr>
<tr>
<td>F_SETFL</td>
<td>Value other than -1.</td>
</tr>
</tbody>
</table>
F_SETLK Value other than -1.
F_SETLKW Value other than -1.

Errors:

EAGAIN The argument cmd is F_SETLK, the type of lock (l_type) is a shared lock (F_RDLCK), and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

EBADF The fildes argument isn’t a valid file descriptor.
The argument cmd is F_SETLK or F_SETLKW, the type of lock (l_type) is a shared lock (F_RDLCK), and fildes isn’t a valid file descriptor open for reading.
The argument cmd is F_SETLK or F_SETLKW, the type of lock (l_type) is an exclusive lock (F_WRLCK), and fildes isn’t a valid file descriptor open for writing.

EDEADLK The argument cmd is F_SETLKW, and a deadlock condition was detected.

EINTR The argument cmd is F_SETLKW, and the function was interrupted by a signal.

EINVAL The argument cmd is F_DUPFD, and the third argument is negative, or greater than the configured number of maximum open file descriptors per process.
The argument cmd is F_GETLK, F_SETLK or F_SETLKW, and the data arg isn’t valid, or fildes refers to a file that doesn’t support locking.
EMFILE The argument cmd is F_DUPFD, and the process has no unused file descriptors, or no file descriptors greater than or equal to arg are available.

ENOLCK The argument cmd is F_SETLK or F_SETLKW, and satisfying the lock or unlock request causes the number of lock regions in the system to exceed the system-imposed limit.

EOVERFLOW One of the values to be returned can’t be represented correctly.

Examples:
/*
 * This program makes "stdout" synchronous
 * to guarantee the data is recoverable
 * (if it’s redirected to a file).
 */
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int flags, retval;

    flags = fcntl( STDOUT_FILENO, F_GETFL );

    flags |= O_DSYNC;

    retval = fcntl( STDOUT_FILENO, F_SETFL, flags );
    if( retval == -1 ) {
        printf( "error setting stdout flags\n" );
        return EXIT_FAILURE;
    }

    printf( "hello QNX world\n" );

    return EXIT_SUCCESS;
}
### Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

### Caveats:

The `fcntl()` function may be a cancellation point in the case of `F_DUPFD` (when dupping across the network), `F_GETFD`, and `F_SETFD`.

### See also:

`close()`, `dup()`, `dup2()`, `execle()`, `execle()`, `execlp()`, `execlpe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `open()`
Synopsis:

```c
#include <unistd.h>

int fdatasync( int filedes );
```

Arguments:

`filedes` The descriptor of the file that you want to synchronize.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fdatasync()` function forces all queued I/O operations for the file specified by the `filedes` file descriptor to finish, synchronizing the file’s data.

This function is similar to `fsync()`, except that `fsync()` also guarantees the integrity of file information, such as access and modification times.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).

Errors:

- `EBADF` The specified `filedes` isn’t a valid file descriptor open for writing.
- `EINVAL` The implementation doesn’t support synchronized I/O for the given file.
- `ENOSYS` The `fdatasync()` function isn’t supported for the filesystem specified by `filedes`. 
Classification:

POSIX 1003.1 SIO

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

aio_fsync(), close(), fcntl(), fsync(), open(), read(), sync(), write()
Synopsis:

```c
#include <stdio.h>

FILE* fdopen( int filedes,
    const char* mode );
```

Arguments:

- `fdopen` The file descriptor that you want to associate with a stream.
- `mode` The mode specified when `fdopen` was originally opened. For information, see `fopen()`, except modes beginning with `w` don’t cause truncation of the file.

Library:

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fdopen()` function associates a stream with the file descriptor `fdopen`, which represents an opened file or device.

The `fdopen()` function preserves the offset maximum previously set for the open file description corresponding to `fdopen`.

Returns:

A file stream for success, or NULL if an error occurs (`errno` is set).

Errors:

- `EBADF` The `fdopen` argument isn’t a valid file descriptor.
- `EINVAL` The `mode` argument isn’t a valid mode.
fdopen()

EMFILE Too many file descriptors are currently in use by this process.
ENOMEM There isn’t enough memory for the FILE structure.

Examples:
```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes;
    FILE *fp;
    
    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        fp = fdopen( filedes , "r" );
        if( fp != NULL ) {
            /* Also closes the underlying FD, filedes. */
            fclose( fp );
        }
    }
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

creat(), dup(), dup2(), errno, fcntl(), fopen(), freopen(), open(), pipe(), sopen()
**feof()**

Test a stream’s end-of-file flag

**Synopsis:**

```c
#include <stdio.h>

int feof( FILE* fp );
```

**Arguments:**

*fp*  The stream you want to test.

**Library:**

*libc*

Use the `-l c` option to *qcc* to link against this library. This library is usually included automatically.

**Description:**

The `feof()` function tests the end-of-file flag for the stream specified by *fp*.

Because the end-of-file flag is set when an input operation attempts to read past the end-of-file, the `feof()` function detects the end-of-file only after an attempt is made to read beyond the end-of-file. Thus, if a file contains 10 lines, the `feof()` won’t detect the end-of-file after the tenth line is read; it will detect the end-of-file on the next read operation.

**Returns:**

0 if the end-of-file flag isn’t set, or nonzero if the end-of-file flag is set.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

void process_record( char *buf )
{
    printf( "%s\n", buf );
}

int main( void )
{
```
FILE *fp;
char buffer[100];

fp = fopen( "file", "r" );
fgets( buffer, sizeof( buffer ), fp );
while( !feof( fp ) ) {
    process_record( buffer );
    fgets( buffer, sizeof( buffer ), fp );
}
fclose( fp );

return EXIT_SUCCESS;

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

clearerr(), ferror(), fgetc(), fgetchar(), fgets(), fgetwc(), fgetws(),
fpopen(), freopen(), getc(), getc_unlocked(), getchar(),
getchar_unlocked(), gets(), getw(), getwc(), getwchar(), perror(),
read()
ferror()

Test a stream’s error flag

Synopsis:

```c
#include <stdio.h>

int ferror( FILE* fp );
```

Arguments:

- `fp` The stream whose error flag you want to test.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ferror()` function tests the error flag for the stream specified by `fp`.

Returns:

- 0 if the error flag isn’t set, or nonzero if the error flag is set.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        c = fgetc( fp );
        if( ferror( fp ) ) {
            printf( "Error reading file\n" );
        }
    }
    fclose( fp );

    return EXIT_SUCCESS;
}
```
ferror()

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

clearerr(), feof(), fgetc(), fgetchar(), fgets(), fgetwc(), fgetws(), getc(), getc_unlocked(), getchar(), getchar_unlocked(), gets(), getw(), getwc(), getwchar(), perror(), strerror()
**Synopsis:**

```c
#include <stdio.h>

int fflush( FILE* fp );
```

**Arguments:**

- `fp` is the NULL, or the stream whose buffers you want to flush.

**Library:**

- **libc**
  
  Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

If the stream specified by `fp` is open for output or update, the `fflush()` function causes any buffered (see `setvbuf()`) but unwritten data to be written to the file descriptor associated with the stream (see `fileno()`).

If the file specified by `fp` is open for input or update, the `fflush()` function undoes the effect of any preceding `ungetc` operation on the stream.

If `fp` is NULL, all open streams are flushed.

**Returns:**

- 0 for success, or EOF if an error occurs (errno is set).

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "Press Enter to continue..." );
    fflush( stdout );
    getchar();

    return EXIT_SUCCESS;
}
```
}\

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `fgetc()`, `fgets()`, `fileno()`, `flushall()`, `fopen()`, `getc()`, `gets()`, `setbuf()`, `setvbuf()`, `ungetc()`
**ffs()**

Find the first bit set in a bit string

**Synopsis:**

```c
#include <strings.h>

int ffs( int value );
```

**Arguments:**

`value` The bit string.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ffs()` function finds the first bit set in `value` and returns the index of that bit. Bits are numbered starting from 1, starting at the rightmost bit.

**Returns:**

The index of the first bit set, or 0 if `value` is zero.

**Classification:**

POSIX 1003.1 XSI

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Synopsis:
#include <stdio.h>

int fgetc( FILE* fp );

Arguments:

fp The stream from which you want to read a character.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The fgetc() function reads the next character from the stream specified by fp.

Returns:
The next character from fp, cast as (int)(unsigned char), or EOF if end-of-file has been reached or if an error occurs (errno is set).

Use feof() or ferror() to distinguish an end-of-file condition from an error.

Examples:
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF ) {

September 10, 2007 C Library — F to H 673
fgetc()

```c
fputc( c, stdout );
}
fclose( fp );

return EXIT_SUCCESS;
}
return EXIT_FAILURE;
```

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `feof()`, `ferror()`, `fgetchar()`, `fgets()`, `fopen()`, `fputc()`, `getc()`, `gets()`, `ungetc()`
Synopsis:
```c
#include <stdio.h>

int fgetchar( void );
```

Library:
```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fgetchar()` function is the same as `fgetc()` with an argument of `stdin`.

Returns:

The next character from `stdin`, cast as `(int)(unsigned char)`, EOF if end-of-file has been reached on `stdin` or if an error occurs (`errno` is set).

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

Examples:
```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = freopen( "file", "r", stdin );
    if( fp != NULL )
    {
        while( (c = fgetchar()) != EOF )
        {
            fputc(c);
        }
    }
    fclose( fp );
}
fgetchar() © 2007, QNX Software Systems GmbH & Co. KG.

    return EXIT_SUCCESS;
  }

    return EXIT_FAILURE;
  }

Classification:

QNX 4

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, feof(), ferror(), fgetc(), fputchar(), getc(), getchar()
fgetpos()

Get the current position of a stream

Synopsis:

```c
#include <stdio.h>

int fgetpos( FILE* fp,
             fpos_t* pos );
```

Arguments:

- `fp` The stream whose position you want to determine.
- `pos` A pointer to a `fpos_t` object where the function can store the position.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fgetpos()` function stores the current position of the stream `fp` in the `fpos_t` object specified by `pos`.

You can use the value stored in `pos` in a call to `fsetpos()` if you want to reposition the file to the position at the time of the `fgetpos()` call.

Returns:

- `0` for success, or nonzero if an error occurs (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    fpos_t position;
    char buffer[80];
```
fp = fopen( "file", "r" );
if( fp != NULL ) {
    fgetpos( fp, &position ); /* get position */
    fgets( buffer, 80, fp ); /* read record */
    fsetpos( fp, &position ); /* set position */
    fgets( buffer, 80, fp ); /* read same record */
    fclose( fp );

    return EXIT_SUCCESS;
}

return EXIT_FAILURE;

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fopen(), fseek(), fsetpos(), ftell()
fgets()

Read a string of characters from a stream

Synopsis:
```c
#include <stdio.h>

char* fgets( char* buf,
            size_t n,
            FILE* fp);
```

Arguments:
- `buf` A pointer to a buffer in which `fgets()` can store the characters that it reads.
- `n` The maximum number of characters to read.
- `fp` The stream from which to read the characters.

Library:
- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `fgets()` function reads a string of characters from the stream specified by `fp`, and stores them in the array specified by `buf`.

It stops reading characters when:
- the end-of-file is reached
  Or:
- a newline (`'\n'`) character is read
  Or:
- `n-1` characters have been read.

The newline character isn’t discarded. A null character is placed immediately after the last character read into the array.
fgets()

Don’t assume that there’s a newline character in every string that you read with fgets(). A newline character isn’t present if there are more than \( n-1 \) characters before the newline.

Also, a newline character might not appear as the last character in a file when the end-of-file is reached.

Returns:

The same pointer as buf, or NULL if the stream is at the end-of-file or an error occurs (errno is set).

Use feof() or ferror() to distinguish an end-of-file condition from an error.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  FILE *fp;
  char buffer[80];

  fp = fopen( "file", "r" );
  if( fp != NULL ) {
    while( fgets( buffer, 80, fp ) != NULL ) {
      fputs( buffer, stdout );
    }
    fclose( fp );
    return EXIT_SUCCESS;
  }
  return EXIT_FAILURE;
}
```
fgets()

Classification:

ANSI, POSIX 1003.1

See also:

errno, feof(), ferror(), fopen(), fputs(), getc(), gets(), fgetc()
fgetspent()  © 2007, QNX Software Systems GmbH & Co. KG.

Get an entry from the shadow password database

Synopsis:

```c
#include <sys/types.h>
#include <shadow.h>

struct spwd* fgetspent( FILE* f);
```

Arguments:

- `f` The stream from which to read the shadow password database.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fgetspent()` works like the `getspent()` function but it assumes that it’s reading from a file formatted like a shadow password database file. This function uses a static buffer that’s overwritten by each call.

The `fgetspent()`, `getspent()`, and `getspnam()` functions share the same static buffer.

Returns:

A pointer to an object of type `struct spwd` containing the next entry from the password database. For more information about this structure, see `putspent()`.

Errors:

The `fgetspent()` function uses the following functions, and as a result `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * This program loops, reading a entries from a file
 * (which is formatted in the shadow password way)
 * reading the next shadow password entry.
 * For example this file /etc/shadow
 */

int main( int argc, char** argv )
{
    FILE* fp;
    struct spwd* sp;
    if (argc < 2) {
        printf("%s filename \n", argv[0]);
        return(EXIT_FAILURE);
    }
    if (!(fp = fopen(argv[1], "r"))) {
        fprintf(stderr, "Can't open file %s \n", argv[1]);
        return(EXIT_FAILURE);
    }
    while( (sp = fgetspent(fp)) != (struct spwd *) 0 ) {
        printf( "Username: %s\n", sp->sp_namp );
        printf( "Password: %s\n", sp->sp_pwdp );
    }
    fclose(fp);
    return( EXIT_SUCCESS );
}
```
Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

errno, getgrent(), getlogin(), getpwnam(), getpwuid(), getspent(),
getspnam(), putspent()
synopsis:

```c
#include <wchar.h>

wint_t fgetwc( FILE * fp );
```

arguments:

- `fp` The stream from which you want to read a character.

library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

description:

The `fgetwc()` function reads the next wide character from the stream specified by `fp`.

returns:

The next character from `fp`, cast as `(wint_t)(wchar_t)`, or WEOF if end-of-file has been reached or if an error occurs (`errno` is set).

- Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

errors:

- **EAGAIN** The O_NONBLOCK flag is set for `fp` and would have been blocked by this operation.
- **EBADF** The file descriptor for `fp` isn’t valid for reading.
- **EINTR** A signal terminated the read operation; no data was transferred.
**fgetwc()**

© 2007, QNX Software Systems GmbH & Co. KG.

- **EIO**: Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.
- **EOVERFLOW**: Cannot read at or beyond the offset maximum for this stream.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `feof()`, `ferror()`, `fputwc()`

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter
fgetws()

Read a string of wide characters from a stream

Synopsis:

```c
#include <wchar.h>

wchar_t * fgetws( wchar_t * buf,
                  int n,
                  FILE * fp );
```

Arguments:

- `buf` A pointer to a buffer in which `fgetws()` can store the wide characters that it reads.
- `n` The maximum number of characters to read.
- `fp` The stream from which to read the characters.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fgetws()` function reads a string of wide characters from the stream specified by `fp`, and stores them in the array specified by `buf`.

It stops reading wide characters when one of the following occurs:

- The end-of-file is reached.
- A newline (‘\n’) character is read.
- `n`-1 characters have been read.

The `fgetws()` function places a NUL at the end of the string.
Don’t assume all strings have newline characters. A newline character isn’t present when more than \( n-1 \) characters occur before the newline.

Also, a newline character might not appear as the last character in a file when the end-of-file is reached.

**Returns:**

- **NULL** Failure; the stream is at the end-of-file or an error occurred (\( \text{errno} \) is set).
- **buf** Success.

Use \( \text{feof()} \) or \( \text{ferror()} \) to distinguish an end-of-file condition from an error.

**Errors:**

- **EAGAIN** The \( \text{O\_NONBLOCK} \) flag is set for \( fp \) and would have been blocked by this operation.
- **EBADF** The file descriptor for \( fp \) isn’t valid for reading.
- **EINTR** A signal terminated the read operation; no data was transferred.
- **EIO** Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.
- **EOVERFLOW** Cannot read at or beyond the offset maximum for this stream.

**Classification:**

- ANSI, POSIX 1003.1
fgetws()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, feof(), ferror(), fputws()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
fileno() — Return the file descriptor for a stream

Synopsis:

```c
#include <stdio.h>

int fileno( FILE * stream );
```

Arguments:

- `stream` The stream whose file descriptor you want to find.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fileno()` function returns the file descriptor for the specified file `stream`. This file descriptor can be used in POSIX input/output calls anywhere the value returned by `open()` can be used.

To associate a stream with a file descriptor, call `fdopen()`.

In QNX Neutrino, the file descriptor is also the connection ID (`coid`) used by various Neutrino-specific functions.

The following symbolic values in `<unistd.h>` define the file descriptors associated with the C language `stdin`, `stdout`, and `stderr` streams:

- `STDIN_FILENO` Standard input file number, `stdin` (0)
- `STDOUT_FILENO` Standard output file number, `stdout` (1)
- `STDERR_FILENO` Standard error file number, `stderr` (2)
fileno()

Returns:
A file descriptor, or -1 if an error occurs (errno is set).

Examples:
```c
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    FILE *stream;

    stream = fopen( "file", "r" );
    if( stream != NULL ) {
        printf( "File number is %d\n", fileno( stream ) );
        fclose( stream );
        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Produces output similar to:

File number is 7.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

errno, fdopen(), fopen(), open()
finite(), finitef()
Determine if a number is finite

Synopsis:

```c
#include <math.h>

int finite ( double x );
int finitef ( float x );
```

Arguments:

- `x`: The number you want to test.

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `finite()` and `finitef()` functions determine if `x` is finite.

Returns:

- True (1) The value of `x` is finite.
- False (≠ 1) The value of `x` is infinity or NAN.

Examples:

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;
    a = 2;
    b = -0.5;
    c = NAN;
    fp_exception_mask(_FP_EXC_DIVZERO, 1);
    d = 1.0/0.0;
```
finite(), finitef()

```c
printf("%f is %s \n", a, (finite(a)) ? "finite" : "not-finite");
printf("%f is %s \n", b, (finite(b)) ? "finite" : "not-finite");
printf("%f is %s \n", c, (finite(c)) ? "finite" : "not-finite");
printf("%f is %s \n", d, (finite(d)) ? "finite" : "not-finite");
return(0);
```

produces the output:

```
2.000000 is finite
-0.500000 is finite
NAN is not-finite
Inf is not-finite
```

**Classification:**

Unix

**Safety**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

isinf(), isnan()
**flink()**

Assign a pathname to a file descriptor

**Synopsis:**

```c
#include <unistd.h>

int flink( int fd, const char *path );
```

**Arguments:**

- `fd`  
  The file descriptor.

- `path`  
  The path you want to associate with the file descriptor.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `flink()` function assigns the pathname, `path`, to the file associated with the file descriptor, `fd`.

**Returns:**

- `0`  
  Success.

- `-1`  
  An error occurred; `errno` is set.

**Errors:**

- **EACCES**  
  A component of either path prefix denies search permission, or the link named by `path` is in a directory with a mode that denies write permission.

- **EBADF**  
  The file descriptor `fd` is invalid.

- **EEXIST**  
  The link named by `path` already exists.

- **ELOOP**  
  Too many levels of symbolic links or prefixes.
flink()

EMLINK  The number of links to the file would exceed LINK_MAX.

ENAMETOOLONG  The length of the path string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT  This error code can mean the following:
  - A component of either path prefix doesn’t exist.
  - The path points to an empty string.

ENOSPC  The directory that would contain the link can’t be extended.

ENOSYS  The flink() function isn’t implemented for the filesystem specified in path.

ENOTDIR  A component of either path prefix isn’t a directory.

EROFS  The requested link requires writing in a directory on a read-only file system.

EXDEV  The link named by path is on a different logical disk.

Classification:
  Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

link()
flock()

Apply or remove an advisory lock on an open file

Synopsis:

```c
#include <fcntl.h>

int flock( int filedes,
           int operation );
```

Arguments:

<table>
<thead>
<tr>
<th>filedes</th>
<th>The file descriptor of an open file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>What you want to do to the file; see below.</td>
</tr>
</tbody>
</table>

Library:

<table>
<thead>
<tr>
<th>libc</th>
</tr>
</thead>
</table>

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `flock()` function applies or removes an advisory lock on the file associated with the open file descriptor `filedes`. To establish a lock with this function, open with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

A lock is applied by specifying one of the following values for `operation`:

- `LOCK_EX`  Exclusive lock.
- `LOCK_NB`  Don’t block when locking. This may be ORed with `LOCK_EX` or `LOCK_SH` to give nonblocking behavior.
- `LOCK_SH`  Shared lock.
- `LOCK_UN`  Unlock an existing lock operation.
Advisory locks allow cooperating processes to perform consistent operations on files, but they don’t guarantee consistency.

The locking mechanism allows two types of locks: shared and exclusive. At any time, multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be upgraded to an exclusive lock, and vice versa, by specifying the appropriate lock type. The previous lock is released and a new lock is applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that’s already locked causes the caller to be blocked until the lock may be acquired. If you don’t want the caller to be blocked, you can specify LOCK_NB in the operation to fail the call (errno is set to EWOULDBLOCK).

Locks are applied to files, not file descriptors. That is, file descriptors duplicated through dup() or fork() don’t result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent loses its lock.

Processes blocked awaiting a lock may be awakened by signals.

**Returns:**

0    The operation was successful.

-1    An error occurred (errno is set).

**Errors:**

EBADF    Invalid descriptor, filedes.

EINVAL    The argument operation doesn’t include one of LOCK_EX, LOCK_SH, or LOCK_UN.

ENOMEM    The system can’t allocate sufficient memory to store lock resources.
flock()

EOPNOTSUPP
   The *filedes* argument refers to an object other than a file.

EWOULDBLOCK
   The file is locked and LOCK_NB was specified.

Classification:
   Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

See also:
   *fcntl(), lockf(), open()*
flockfile()
Acquire ownership of a file

Synopsis:
#include <stdio.h>

void flockfile( FILE* file );

Arguments:

file A pointer to the FILE object for the file you want to lock.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The flockfile() function provides for explicit application-level locking of stdio (FILE) objects. This function can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The flockfile() function is used by a thread to acquire ownership of a FILE.

The implementation acts as if there is a lock count associated with each FILE. This count is implicitly initialized to zero when the FILE is created. The FILE object is unlocked when the count is zero. When the count is positive, a single thread owns the FILE. When the flockfile() function is called, if the count is zero or if the count is positive and the caller owns the FILE, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero.

Classification:

POSIX 1003.1 TSF
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`, `putchar_unlocked()`
Round down a value to the next integer

Synopsis:

```c
#include <math.h>

double floor( double x );

float floorf( float x );
```

Arguments:

`x`  The value you want to round.

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the largest integer ≤ x (rounding towards the “floor”).

Returns:

The largest integer ≤ x.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", floor( -3.14 ) );
    printf( "%f\n", floor( -3. ) );
    printf( "%f\n", floor( 0. ) );
    printf( "%f\n", floor( 3. ) );
    printf( "%f\n", floor( 6.28 ) );
    printf( "%f\n", floor( -1.57 ) );
    printf( "%f\n", floor( 1.57 ) );
    printf( "%f\n", floor( 1.99 ) );
}```


```c
printf( "%f\n", floor( 3.14 ) );
printf( "%f\n", floor( 3. ) );

return EXIT_SUCCESS;
}
```

produces the output:

```
-4.000000
-3.000000
0.000000
3.000000
3.000000
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`ceil(), fmod()`
Synopsis:

```c
#include <stdio.h>

int flushall( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `flushall()` function flushes all buffers associated with open input/output streams. A subsequent read operation on an input stream reads new data from the associated stream.

Calling the `flushall()` function is equivalent to calling `fflush()` for all open streams.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).

Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
flushall()

Caveats:

The QNX 4 version of this function returns the number of streams flushed.

See also:

errno, fopen(), flush()
Synopsis:

```c
#include <math.h>

double fmod( double x, 
        double y );

float fmodf( float x, 
        float y );
```

Arguments:

- `x`: An arbitrary number.
- `y`: The modulus.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

The `fmod()` and `fmodf()` functions compute the floating-point residue of `x (mod y)`, which is the remainder of `x ÷ y`, even if the quotient `x ÷ y` isn’t representable.

Returns:

The residue, `x - (i × y)`, for some integer `i` such that, if `y` is nonzero, the result has the same sign as `x` and a magnitude less than the magnitude of `y`.

If `y` is zero, the function returns 0.
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set \texttt{errno} to 0, call the function, and then check \texttt{errno} again. These functions don’t change \texttt{errno} if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", fmod( 4.5, 2.0 ) );
    printf( "%f\n", fmod( -4.5, 2.0 ) );
    printf( "%f\n", fmod( 4.5, -2.0 ) );
    printf( "%f\n", fmod( -4.5, -2.0 ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0.500000
-0.500000
0.500000
-0.500000
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

ceil(), div(), fabs(), floor()
fnmatch()

Check to see if a file or path name matches a pattern

Synopsis:

```c
#include <fnmatch.h>

int fnmatch(const char* pat, const char* str, int flags);
```

Arguments:

- `pat` The pattern to match; see “Pattern Matching Special Characters,” below.
- `str` The string to match against the pattern.
- `flags` Flags that modify interpretation of `pat` and `str`; a bitwise inclusive OR of these bits:

  - **FNM_PATHNAME**
    If this is set, a slash character in `str` is explicitly matched by a slash in `pat`; it isn’t matched by either the asterisk or question mark special characters, or by a bracket expression.

  - **FNM_PERIOD**
    If this is set, a leading period in `str` matches a period in `pat`, where the definition of “leading” depends on FNM_PATHNAME:
    - If FNM_PATHNAME is set, a period is leading if it’s the first character in `str`, or if it immediately follows a slash.
    - If FNM_PATHNAME isn’t set, a period is leading only if it’s the first character in `str`.

  - **FNM_QUOTE**
    If this isn’t set, a backslash (\) in `pat` followed by another character matches that second character. If FNM_QUOTE is set, a backslash is treated as an ordinary character.
Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fnmatch()` function checks the file or path name specified by the `str` argument to see if it matches the pattern specified by the `pat` argument.

Pattern Matching Special Characters

A pattern-matching special character that is quoted is a pattern that matches the special character itself. When not quoted, such special characters have special meaning in the specification of patterns. The pattern-matching special characters and the contexts in which they have their special meaning are as follows:

- `?` Matches any printable or nonprintable collating element except `<newline>`.
- `*` Matches any string, including the null string.
- `[bracket_expr]` Matches a single collating element as per Regular Expression Bracket Expressions (1003.2 2.9.1.2) except that:
  - The exclamation point character (!) replaces the circumflex character (`) in its role as a nonmatching list in the regular expression notation.
  - The backslash is used as an escape character within bracket expressions.

The `?`, `*` and `[` characters aren’t special when used inside a bracket expression.

The concatenation of patterns matching a single character is a valid pattern that matches the concatenation of the single characters or
collating elements matched by each of the concatenated patterns. For example, the pattern `a[bc]` matches the strings `ab` and `ac`.

The concatenation of one or more patterns matching a single character with one or more asterisks (*) is a valid pattern. In such patterns, each asterisk matches a string of zero or more characters, up to the first character that matches the character following the asterisk in the pattern. For example, the pattern `a*d` matches the strings `ad`, `abd`, and `abcd`, but not the string `abc`.

When an asterisk is the first or last character in a pattern, it matches zero or more characters that precede or follow the characters matched by the remainder of the pattern. For example, the pattern `a*d*` matches the strings `ad`, `abcd`, `abcdef`, `aaaad` and `addddd`; the pattern `*a*d` matches the strings `ad`, `abcd`, `efabcd`, `aaaad` and `addddd`.

Returns:

0 The `str` argument matches the pattern specified by `pat`.

Nonzero The `str` argument doesn’t match the pattern specified by `pat`.

Examples:

```c
/*
 * The following example accepts a set of patterns
 * for filenames as argv[1..argc]. It reads lines
 * from standard input, and outputs the lines that
 * match any of the patterns.
 */
#include <stdio.h>
#include <fnmatch.h>
#include <stdlib.h>
#include <limits.h>
int main( int argc, char **argv )
{
    int i;
    char buffer[PATH_MAX+1];
    while( gets( buffer ) ) {
        for( i = 0; i < argc; i++ ) {
            if( fnmatch( argv[i], buffer, 0 ) == 0 ) {
                puts( buffer );
            }
        }
    }
}
```
fnmatch()

```c
    break;
    }
  
  exit( EXIT_SUCCESS );
```

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`regcomp()`
Synopsis:

```c
#include <stdio.h>

FILE * fopen( const char * filename,
              const char * mode );

FILE * fopen64( const char * filename,
                const char * mode );
```

Arguments:

- `filename` The name of the file that you want to open.
- `mode` The access mode; see below.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fopen()` and `fopen64()` functions open a file stream for the file specified by `filename`. The `mode` string begins with one of the following sequences:

- `a` Append: create a new file or open the file for writing at its end.
- `a+` Append: open the file or create it for update, writing at end-of-file; use the default file translation.
- `r` Open the file for reading.
- `r+` Open the file for update (reading and/or writing); use the default file translation.
- `w` Create the file for writing, or truncate it to zero length.
Create the file for update, or truncate it to zero length; use the default file translation.

You can add the letter `b` to the end of any of the above sequences to indicate that the file is (or must be) a binary file (this is an ANSI requirement for portability to systems that make a distinction between text and binary files, such as DOS). Under QNX Neutrino, there’s no difference between text files and binary files.

- Opening a file in read mode (`r` in the `mode`) fails if the file doesn’t exist or can’t be read.

- Opening a file in append mode (`a` in the `mode`) causes all subsequent writes to the file to be forced to the current end-of-file, regardless of previous calls to the `fseek()` function.

- When a file is opened with update mode (`+` in the `mode`), both input and output may be performed on the associated stream.

When using a stream in update mode, writing can’t be followed by reading without an intervening call to `fflush()`, or to a file-positioning function (`fseek()`, `fsetpos()` or `rewind()`). Similarly, reading can’t be followed by writing without an intervening call to a file-positioning function, unless the read resulted in end-of-file.

The largest value that can be represented correctly in an object of type `off_t` shall be established as the offset maximum in the open file description.

**Returns:**

A pointer to a file stream for success, or NULL if an error occurs (`errno` is set).

**Errors:**

- **EACCES** Search permission is denied on a component of the `filename` prefix, or the file exists and the
permissions specified by *mode* are denied, or the file doesn’t exist and write permission is denied for the parent directory of the file to be created.

**EBADFSYS** While attempting to open the named file, either the file itself or a component of the *filename* prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You’ll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

**EBUSY** File access was denied due to a conflicting open (see *sopen()*).

**EINTR** The *fopen()* operation was interrupted by a signal.

**EINVAL** The value of the *mode* argument is not valid.

**EISDIR** The named file is a directory, and the *mode* argument specifies write-only or read/write access.

**ELOOP** Too many levels of symbolic links or prefixes.

**EMFILE** Too many file descriptors are currently in use by this process.

**ENAMETOOLONG**

The length of the *filename* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

**ENFILE** Too many files are currently open in the system.

**ENOENT** Either the named file or the *filename* prefix doesn’t exist, or the *filename* argument points to an empty string.

**ENOMEM** There isn’t enough memory for the *FILE* structure.
ENOSPC The directory or filesystem that would contain the new file can’t be extended.

ENOSYS The fopen() function isn’t implemented for the filesystem specified in filename.

ENOTDIR A component of the filename prefix isn’t a directory.

ENXIO The media associated with the file has been removed (e.g. CD, floppy).

EOVERFLOW The named file is a regular file and the size of the file can’t be represented correctly in an object of type off_t.

EROFS The named file resides on a read-only filesystem.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;

    fp = fopen( "report.dat", "r" );
    if( fp != NULL ) {
        /* rest of code goes here */
        fclose( fp );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}
```

Classification:

fopen() is ANSI, POSIX 1003.1; fopen64() is Large-file support
fopen(), fopen64()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fclose(), fcloseall(), fdopen(), freopen(), freopen64()
Synopsis:

```c
#include <sys/types.h>
#include <process.h>

pid_t fork( void );
```

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fork()` function creates a new process. The new process (child process) is an exact copy of the calling process (parent process), except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (which is the process ID of the calling process).
- The child process has its own copy of the parent’s file descriptors. Each of the child’s file descriptors refers to the same open file description with the corresponding file descriptor of the parent.
- The child process has its own copy of the parent’s open directory streams.
- The child process’s values of `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` are set to zero.
- File locks previously set by the parent aren’t inherited by the child.
- Pending alarms are cleared for the child process.
- The set of signals pending for the child process is initialized to the empty set.
Returns:

A value of zero to the child process; and the process ID of the child process to the parent process. Both processes continue to execute from the `fork()` function. If an error occurs, `fork()` returns -1 to the parent and sets `errno`.

Errors:

- **EAGAIN** Insufficient resources are available to create the child process. For example, you might have exceeded the maximum number of processes permitted; see the `RLIMIT_NPROC` resource for `getrlimit()`.
- **ENOMEM** The process requires more memory than the system is able to supply.
- **ENOSYS** The `fork()` function isn’t implemented for this memory protection model. See also “Caveats,” below.

Examples:

```c
/*
 * This program executes the program and arguments
 * specified by argv[1..argc]. The standard input
 * of the executed program is converted to upper
 * case.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <process.h>
#include <sys/wait.h>

int main( int argc, char **argv )
{
    pid_t pid;
    pid_t wpid;
    int   fd[2];
    char  buffer[80];
    int   i, len;
    int   status;

    if( pipe( fd ) == -1 ) {
```
```c
fork()

    perror( "pipe" );
    return EXIT_FAILURE;

}

if( ( pid = fork() ) == -1 ) {
    perror( "fork" );
    return EXIT_FAILURE;
}

if( pid == 0 ) {
    /* This is the child process.
     * Move read end of the pipe to stdin ( 0 ),
     * close any extraneous file descriptors,
     * then use exec to 'become' the command.
     */
    dup2( fd[0], 0 );
    close( fd[1] );
    execvp( argv[1], argv+1 );

    /* This can only happen if exec fails; print message
     * and exit.
     */
    perror( argv[1] );
    return EXIT_FAILURE;
} else {
    /* This is the parent process.
     * Remove extraneous file descriptors,
     * read descriptor 0, write into pipe,
     * close pipe, and wait for child to die.
     */
    close( fd[0] );
    while( ( len = read( 0, buffer, sizeof( buffer ) ) ) > 0 ) {
        for( i = 0; i < len; i++ ) {
            if( isupper( buffer[i] ) )
                buffer[i] = tolower( buffer[i] );
        }
        write( fd[1], buffer, len );
    }
    close( fd[1] );
    do {
        wpid = waitpid( pid, &status, 0 );
    } while( WIFEXITED( status ) == 0 );
    return WEXITSTATUS( status );
}
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Currently, `fork()` is supported only in single-threaded applications. If you create a thread and then call `fork()`, the function returns -1 and sets `errno` to ENOSYS.

See also:

`errno`, `execl()`, `execlp()`, `execle()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`,
forkpty()

Create a new process operating in a pseudo-tty

Synopsis:

```c
#include <unix.h>

pid_t forkpty( int *amaster,
               char *name,
               struct termios *termp,
               struct winsize *winp );
```

Arguments:

- **amaster**: A pointer to a location where `forkpty()` can store the file descriptor of the master side of the pseudo-tty.
- **name**: NULL, or a pointer to a buffer where `forkpty()` can store the filename of the slave side of the pseudo-tty.
- **termp**: NULL, or a pointer to a `termios` structure that describes the terminal’s control attributes to apply to the slave side of the pseudo-tty.
- **winp**: A pointer to a `winsize` structure that defines the window size to use for the slave side of the pseudo-tty.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `forkpty()` function combines `openpty()`, `fork()`, and `login_tty()` to create a new process operating in a pseudo-tty.

This function fails if either `openpty()` or `fork()` fails.
forkpty()

Returns:

0 to the child process, the child’s process ID to the parent, or -1 if an error occurred.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fork(), login_tty(), openpty(), termios
**fp_exception_mask()**

*Get or set the current exception mask*

**Synopsis:**

```c
#include <fpstatus.h>

int fp_exception_mask ( int new_mask,
                        int set);
```

**Arguments:**

- `new_mask` - The new mask to apply. The bits include:
  - `_FP_EXC_INVALID`
  - `_FP_EXC_DIVZERO`
  - `_FP_EXC_OVERFLOW`
  - `_FP_EXC_UNDERFLOW`
  - `_FP_EXC_INEXACT`
  - `_FP_EXC_DENORMAL`

- `set` - A value that indicates what you want the function to do:
  - If `set < 0`, return the current mask. The `new_mask` argument is ignored.
  - If `set = 0`, disable the bits in the exception mask that correspond to the bits set in `new_mask`.
  - If `set > 0`, enable the bits in the exception mask that correspond to the bits set in `new_mask`.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `fp_exception_mask()` function gets or sets the current exception mask, depending on the value of the `set` argument.
Returns:

If \( set < 0 \) The current exception mask.
If \( set \geq 0 \) The previous mask.

This function doesn’t return a special value to indicate that an error occurred. If you want to check for errors, set \( errno \) to 0, call the function, and then check \( errno \) again.

Examples:

```c
#include <fpstatus.h>

int main(int argc, char** argv)
{
    int ret;

    if ((ret = fp_exception_mask(0, -1)) < 0)
        printf("*** Problem retrieving exceptions \n");
    printf("Exceptions Enabled: \n\t");
    if (ret & _FP_EXC_INEXACT)
        printf("Inexact ");
    if (ret & _FP_EXC_DIVZERO)
        printf("DivZero ");
    if (ret & _FP_EXC_UNDERFLOW)
        printf("Underflow ");
    if (ret & _FP_EXC_OVERFLOW)
        printf("Overflow ");
    if (ret & _FP_EXC_INVALID)
        printf("Invalid ");
    printf("\n");

    /* Set the exception mask to enable division by zero errors */
    if ((ret = fp_exception_mask(_FP_EXC_DIVZERO, 1)) < 0)
        printf("*** Problem setting exceptions \n");
    if ((ret = fp_exception_mask(0, -1)) < 0)
        printf("*** Problem retrieving exceptions \n");
    printf("Exceptions Enabled: \n\t");
    if (ret & _FP_EXC_INEXACT)
        printf("Inexact ");
    if (ret & _FP_EXC_DIVZERO)
        printf("DivZero ");
    if (ret & _FP_EXC_UNDERFLOW)
        printf("Underflow ");
    if (ret & _FP_EXC_OVERFLOW)
        printf("Overflow ");
    if (ret & _FP_EXC_INVALID)
printf("Overflow ");
if (ret & _FP_EXC_INVALID)
    printf("Invalid ");
printf("\n");
    return(0);
}

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`fp_exception_value()`, `fp_precision()`, `fp_rounding()`
**fp_exception_value()**

Get the value of the current exception registers

**Synopsis:**

```c
#include <fpstatus.h>

int fp_exception_value( int mask );
```

**Arguments:**

*mask* A mask whose bits indicate which registers you want the value of. The bits include:

- _FP_EXC_INVALID
- _FP_EXC_DIVZERO
- _FP_EXC_OVERFLOW
- _FP_EXC_UNDERFLOW
- _FP_EXC_INEXACT
- _FP_EXC_DENORMAL

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `fp_exception_value()` function gets the value of the current exception registers. Set bits indicate that the exception has signaled, unset bits indicate that the exception hasn’t signaled.

**Returns:**

The value of the current exception registers based on the values from `<fpstatus.h>`.
This function doesn’t return a special value to indicate that an error occurred. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again.

Examples:

```c
#include <fpstatus.h>

int main(int argc, char** argv)
{
    int ret;

    /* Test to see if an operation has set (but not necessarily
    * signaled depending on the exception mask) the
    * division by zero bit:
    */

    if (fp_exception_value(_FP_EXC_DIVZERO) & _FP_EXC_DIVZERO)
        printf("Division by zero has occurred \n");
    else
        printf("Division by zero has not occurred \n");

    return(0);
}
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

\[fp\_precision(), fp\_rounding(), fp\_exception\_mask()\]
fp_precision()
Set or get the current precision

Synopsis:

```c
#include <fpstatus.h>

int fp_precision( int newprecision );
```

Arguments:

- **newprecision**  
The new precision; one of:
  - `< 0` — return the current setting.
  - `_FP_PREC_FLOAT`
  - `_FP_PREC_DOUBLE`
  - `_FP_PREC_EXTENDED`
  - `_FP_PREC_DOUBLE_EXTENDED`

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `fp_precision()` function sets or gets the current floating-point precision, depending on the value of `newprecision`.

Returns:

If `newprecision` is less than 0, the current precision; otherwise, the previous precision.

This function doesn’t return a special value to indicate that an error occurred. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again.
Examples:

```c
#include <fpstatus.h>

int main(int argc, char** argv)
{
    int ret;

    ret = fp_precision(-1);
    printf("Precision: ");
    if (ret == _FP_PREC_FLOAT)
        printf("Float \n");
    else if (ret == _FP_PREC_DOUBLE)
        printf("Double \n");
    else if (ret == _FP_PREC_EXTENDED)
        printf("Extended \n");
    else if (ret == _FP_PREC_DOUBLE_EXTENDED)
        printf("128 Bit \n");
    else if (ret == _FP_PREC_EXTENDED)
        printf("Extended \n");
    else
        printf("Error \n");

    return(0);
}
```

Classification:

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

\[ \text{fp\_exception\_mask()}, \text{fp\_exception\_value()}, \text{fp\_rounding()} \]
Synopsis:

```
#include <fpstatus.h>

int fp_rounding( int newrounding );
```

Arguments:

- `newrounding` The new rounding; one of:
  - `< 0` — return the current setting.
  - `_FP_ROUND_NEAREST`
  - `_FP_ROUND_ZERO`
  - `_FP_ROUND_POSITIVE`
  - `_FP_ROUND_NEGATIVE`

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `fp_rounding()` function sets or gets the current rounding mode, depending on the value of `newrounding`.

Returns:

If `newrounding` is less than 0, the current rounding mode; otherwise, the previous mode.

This function doesn’t return a special value to indicate that an error occurred. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again.
Examples:

```c
#include <fpstatus.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    int ret;
    ret = fp_rounding(-1);
    printf("Rounding mode: ");
    if (ret == _FP_ROUND_NEAREST)
        printf("Nearest \
");
    else if (ret == _FP_ROUND_POSITIVE)
        printf("Positive \
");
    else if (ret == _FP_ROUND_NEGATIVE)
        printf("Negative \
");
    else if (ret == _FP_ROUND_ZERO)
        printf("To Zero \
");
    else
        printf("Error \
");

    return EXIT_SUCCESS;
}
```

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`fp_exception_mask()`, `fp_exception_value()`, `fp_precision()`
fpathconf()  © 2007, QNX Software Systems GmbH & Co. KG.
Return the value of a configurable limit associated with a file

Synopsis:

```c
#include <unistd.h>

long fpathconf( int filedes,
                int name );
```

Arguments:

- `filedes` A file descriptor for the file whose limit you want to check.
- `name` The name of the configurable limit; see below.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fpathconf()` function returns a value of a configurable limit indicated by `name` that’s associated with the file indicated by `filedes`.

Configurable limits are defined in `<confname.h>`, and include at least the following values:

- `_PC_LINK_MAX`
  - Maximum value of a file’s link count.

- `_PC_MAX_CANON`
  - Maximum number of bytes in a terminal’s canonical input buffer (edit buffer).

- `_PC_MAX_INPUT`
  - Maximum number of bytes in a terminal’s raw input buffer.

- `_PC_NAME_MAX`
  - Maximum number of bytes in a file name (not including the terminating null).
_PC_PATH_MAX
    Maximum number of bytes in a pathname (not including the
    terminating null).

_PC_PIPE_BUF
    Maximum number of bytes that can be written atomically when
    writing to a pipe.

_PC_CHOWN_RESTRICTED
    If defined (not -1), indicates that the use of the chown() function
    is restricted to a process with appropriate privileges, and to
    changing the group ID of a file to the effective group ID of the
    process or to one of its supplementary group IDs.

_PC_NO_TRUNC
    If defined (not -1), indicates that the use of pathname
    components longer than the value given by _PC_NAME_MAX
    generates an error.

_PC_VDISABLE
    If defined (not -1), this is the character value that can be used to
    individually disable special control characters in the termios
    control structure.

Returns:

The requested configurable limit, or -1 if an error occurred (errno is
set).

Errors:

EINVAL    The name argument is invalid, or the indicated limit isn’t
          supported for this filedes.

EBADF    The argument filedes is invalid.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    long value;

    value = fpathconf( 0, _PC_MAX_INPUT );
    printf( "Input buffer size is %ld bytes\n", value );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`confstr()`, `pathconf()`, `sysconf()`, `termios`
Synopsis:

```c
#include <stdio.h>

int fprintf( FILE* fp,
             const char* format,
             ... );
```

Arguments:

- `fp` The stream to which you want to send the output.
- `format` A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fprintf()` function writes output to the stream specified by `fp`, under control of the `format` specifier.

Returns:

The number of characters written, or a negative value if an output error occurred (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
char *weekday = { "Saturday" }; char *month = { "April" };

int main( void )
{
    fprintf( stdout, "%s, %s %d, %d
", September 10, 2007 C Library — F to H 739
```
weekday, month, 10, 1999);
    return EXIT_SUCCESS;
}

Produces:

Saturday, April 10, 1999

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fprintf(), printf(), sprintf(), snprintf(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
Synopsis:

```c
#include <stdio.h>

int fputc( int c,
            FILE* fp );
```

Arguments:
- `c`  The character you want to write.
- `fp`  The stream you want to write the character to.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fputc()` function writes the character specified by `c`, cast as `(int)(unsigned char)`, to the stream specified by `fp`.

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF ) {
            fputc( c, stdout );
        }
    }
    fclose( fp );
}
```
fclose(fp);

    return EXIT_SUCCESS;
}

return EXIT_FAILURE;


Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

If c is negative, the value returned by this function isn’t equal to c — unless c is -1 and an error occurred :-)

See also:

errno, fgetc(), fopen(), fprintf(), fputchar(), fputs(), putc(), putchar(), puts()
Synopsis:

```c
#include <stdio.h>

int fputchar( int c );
```

Arguments:

c  The character you want to write.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fputchar()` function writes the character specified by `c`, cast as `(int)(unsigned char)`, to `stdout`. It’s equivalent to `putchar()` and to:

```c
fputc( c, stdout );
```

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        c = fgetc( fp );
        while( c != EOF ) {
```

September 10, 2007 C Library — F to H 743
fputchar()  

```c
fputchar( c );
  c = fgetc( fp );
}  
fclose( fp );
  return EXIT_SUCCESS;
}
return EXIT_FAILURE;
```

**Classification:**

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

If `c` is negative, the value returned by this function isn’t equal to `c` — unless `c` is -1 and an error occurred :-)

**See also:**

`errno, fgetc(), fgetchar(), fprintf(), fputc(), fputs(), putc(), putchar()`
Synopsis:

```c
#include <stdio.h>

int fputs( const char* buf, FILE* fp );
```

Arguments:

- **buf** The string you want to write.
- **fp** The stream you want to write the string to.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fputs()` function writes the character string specified by `buf` to the output stream specified by `fp`.

**Tip:** The terminating NUL character isn’t written.

Returns:

A nonnegative value for success, or EOF if an error occurs (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  FILE *fp_in, *fp_out;
  char buffer[80];
```
fp_in = fopen( "file", "r" );
fp_out = fopen( "outfile", "w" );
if( fp_in != NULL && fp_out != NULL ) {
    while( fgets( buffer, 80, fp_in ) != NULL ) {
        fputs( buffer, fp_out );
    }
    fclose( fp_in );
    fclose( fp_out );
    return EXIT_SUCCESS;
}
return EXIT_FAILURE;

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fgets(), fopen(), fprintf(), fputs(), puts()
fputwc()

Write a wide character to a stream

Synopsis:

#include <wchar.h>

wint_t fputwc( wchar_t wc,
               FILE * fp );

Arguments:

wc  The wide character you want to write.
fp   The stream you want to write the character to.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The fputwc() function writes the wide character specified by wc, cast as (wint_t) (wchar_t), to the stream specified by fp.

Returns:

The wide character written, cast as (wint_t) (wchar_t), or WEOF if an error occurred (errno is set).

If wc exceeds the valid wide-character range, the value returned is the wide character written, not wc.

Errors:

EAGAIN   The O_NONBLOCK flag is set for fp and would have been blocked by this operation.
EBADF    The stream specified by fp isn’t valid for writing.
fputwc()  

The file exceeds the maximum file size, the process’s file size limit, or the function can’t write at or beyond the offset maximum.

A signal terminated the write operation; no data was transferred.

A physical I/O error has occurred or all of the following conditions were met:

- The process is in the background.
- TOSTOP is set.
- The process is blocking/ignoring SIGTTOU.
- The process group is orphaned.

Can’t write to pipe or FIFO because it’s closed; a SIGPIPE signal is also sent to the thread.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fgetwc(), fputws()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```
#include <wchar.h>

int fputws( const wchar_t * ws,  
         FILE * fp );
```

Arguments:

- `buf`  The wide-character string you want to write.
- `fp`  The stream you want to write the string to.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fputws()` function writes the wide-character string specified by `ws` to the output stream specified by `fp`.

- **Tip** The terminating NUL wide character isn’t written.

Returns:

A nonnegative value for success, or WEOF if an error occurs (`errno` is set).

Errors:

- **EAGAIN**  The O_NONBLOCK flag is set for `fp` and would have been blocked by this operation.
- **EBADF**  The stream specified by `fp` isn’t valid for writing.
**fputws()**

The file exceeds the maximum file size, the process’s file size limit, or the function can’t write at or beyond the offset maximum.

**EINTR**
A signal terminated the write operation; no data was transferred.

**EIO**
A physical I/O error has occurred or all of the following conditions were met:
- The process is in the background.
- TOSTOP is set.
- The process is blocking/ignoring SIGTTOU.
- The process group is orphaned.

**EPIPE**
Can’t write to pipe or FIFO because it’s closed; a SIGPIPE signal is also sent to the thread.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `fgetws()`, `fputwc()`

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```c
#include <stdio.h>

size_t fread( void* buf,
              size_t size,
              size_t num,
              FILE* fp );
```

Arguments:

- `buf` A pointer to a buffer where the function can store the elements that it reads.
- `size` The size of each element to read.
- `num` The number of elements to read.
- `fp` The stream from which to read the elements.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fread()` function reads `num` elements of `size` bytes each from the stream specified by `fp` into the buffer specified by `buf`.

Returns:

The number of complete elements successfully read; this value may be less than the requested number of elements.

Use the `feof()` and `ferror()` functions to determine whether the end of the file was encountered or if an input/output error has occurred.
Errors:

If an error occurs, *errno* is set to indicate the type of error.

Examples:

The following example reads a simple student record containing binary data. The student record is described by the `struct student_data` declaration.

```c
#include <stdio.h>
#include <stdlib.h>

struct student_data {
    int student_id;
    unsigned char marks[10];
};

size_t read_data( FILE *fp, struct student_data *p )
{
    return( fread( p, sizeof( struct student_data ), 1, fp ) );
}

int main( void )
{
    FILE *fp;
    struct student_data std;
    int i;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( read_data( fp, &std ) != 0 ) {
            printf( "id=%d 
" , std.student_id );
            for( i = 0; i < 10; i++ ) {
                printf( "%3d 
" , std.marks[ i ] );
            }
            printf( "\n" );
        }
        fclose( fp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```
Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

erro, fopen(), feof(), ferror()
free()

Deallocate a block of memory

Synopsis:

```c
#include <stdlib.h>

void free( void* ptr );
```

Arguments:

- `ptr`: A pointer to the block of memory that you want to free. It’s safe to call `free()` with a NULL pointer.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `free()` function deallocates the memory block specified by `ptr`, which was previously returned by a call to `calloc()`, `malloc()` or `realloc()`.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main( void )
{
    char *buffer;
    buffer = (char *)malloc( 80 );
    if( buffer == NULL ) {
        printf( "Unable to allocate memory\n" );
        return EXIT_FAILURE;
    } else {
        /* rest of code goes here */
        free( buffer ); /* deallocate buffer */
    }
    return EXIT_SUCCESS;
}
```
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

Calling `free()` on a pointer already deallocated by a call to `free()` or `realloc()` could corrupt the memory allocator’s data structures.

See also:

`alloca()`, `calloc()`, `malloc()`, `realloc()`, `sbrk()`
**freeaddrinfo()**  
Free a list of address information structures

### Synopsis:
```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo( struct addrinfo * ai );
```

### Arguments:
- `ai` A pointer to the `addrinfo` structure that’s at the beginning of the list to be freed.

### Library:
- **libsocket**
  Use the `-l socket` option to `qcc` to link against this library.

### Description:
The `freeaddrinfo()` function frees the given list of `addrinfo` structures and the dynamic storage associated with each item in the list.

### Classification:
- POSIX 1003.1

### Safety
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

addrinfo, gai_strerror, getaddrinfo
freeifaddrs()

Free a network interface address

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>

void freeifaddrs( struct ifaddrs * ifap );
```

Arguments:

- `ifap` A pointer to the linked list of `ifaddrs` structures to be freed.

Library:

- **libsocket**
  
  Use the `-l socket` option to `qcc` to link against this library.

Description:

The `freeifaddrs()` function frees the dynamically allocated data returned by `getifaddrs()`.

Classification:

- Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`getifaddrs()`, `ifaddrs`, `ioctl()`, `malloc()`, `socket()`, `sysctl()`
freopen(), freopen64()

Reopen a stream

Synopsis:

```c
#include <stdio.h>

FILE* freopen( const char* filename,
    const char* mode,
    FILE* fp );

FILE* freopen64( const char* filename,
    const char* mode,
    FILE* fp );
```

Arguments:

- `filename` - The name of the file to open.
- `mode` - The mode to use when opening the file. For more information, see `fopen()`.
- `fp` - The stream to associate with the file.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `freopen()` and `freopen64()` functions close the open stream `fp`, open the file specified by `filename`, and associate its stream with `fp`.

The largest value that can be represented correctly in an object of type `off_t` shall be established as the offset maximum in the open file description.

Returns:

A pointer to the newly opened stream, or NULL if an error occurs (`errno` is set).
Errors:

EACCES  Search permission is denied on a component of the
filename prefix, or the file exists and the
permissions specified by mode are denied, or the
file doesn’t exist and write permission is denied for
the parent directory of the file to be created.

EBADFSYS While attempting to open the named file, either the
file itself or a component of the filename prefix
was found to be corrupted. A system failure —
from which no automatic recovery is possible —
occurred while the file was being written to, or
while the directory was being updated. You’ll need
to invoke appropriate systems-administration
procedures to correct this situation before
proceeding.

EBUSY  File access was denied due to a conflicting open
(see sopen()).

EINVAL  The value of the mode argument is not valid.

EISDIR  The named file is a directory, and the mode
argument specifies write-only or read/write access.

ELoop  Too many levels of symbolic links or prefixes.

EMFILE  Too many file descriptors are currently in use by
this process.

ENAMETOOLONG
The length of the filename string exceeds
PATH_MAX, or a pathname component is longer
than NAME_MAX.

ENFILE  Too many files are currently open in the system.
`freopen()`, `freopen64()` © 2007, QNX Software Systems GmbH & Co. KG.

ENOENT Either the named file or the `filename` prefix doesn’t exist, or the `filename` argument points to an empty string.

ENOMEM There is no memory for FILE structure.

ENOSPC The directory or filesystem that would contain the new file can’t be extended.

ENOSYS The `freopen()` function isn’t implemented for the filesystem specified in `filename`.

ENOTDIR A component of the `filename` prefix isn’t a directory.

ENXIO The media associated with the file has been removed (e.g. CD, floppy).

EOVERFLOW The named file is a regular file and the size of the file can’t be represented correctly in an object of type `off_t`.

EROFS The named file resides on a read-only filesystem.

Examples:
```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE* fp;
    int c;

    /* Reopen the stdin stream so it’s reading
     * from "file" instead of standard input.
     */
    fp = freopen( "file", "r", stdin );

    if( fp != NULL ) {
        /* Now we can read from "file" using the
         * stdin functions like fgetchar()...
         */
        while( ( c = fgetchar() ) != EOF ) {
            fputchar( c );
```
```c
}

fclose( fp );
return EXIT_SUCCESS;
}

return EXIT_FAILURE;
```

**Classification:**

`freopen()` is ANSI, POSIX 1003.1; `freopen64()` is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `fclose()`, `fcloseall()`, `fdopen()`, `fopen()`, `fopen64()`
frexp(), frexpf()  © 2007, QNX Software Systems GmbH & Co. KG.

Break a floating-point number into a normalized fraction and an integral power of 2

Synopsis:

```c
#include <math.h>

double frexp( double value,
             int* exp );

float frexpf( float value,
             int* exp );
```

Arguments:

- `value` The value you want to break into a normalized fraction.
- `exp` A pointer to a location where the function can store the integral power of 2.

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions break a floating-point number into a normalized fraction and an integral power of 2. It stores the integral power of 2 in the `int` pointed to by `exp`.

Returns:

`x`, such that `x` is a `double` with magnitude in the interval [0.5, 1] or 0, and `value` equals `x` times 2 raised to the power `exp`. If `value` is 0, then both parts of the result are 0.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
```
int expon;
double value;

value = frexp( 4.25, &expon );
printf( "%f %d\n", value, expon );
value = frexp( -4.25, &expon );
printf( "%f %d\n", value, expon );

return EXIT_SUCCESS;
}

produces the output:

0.531250 3
-0.531250 3

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ldexp(), modf()
**Synopsis:**
```c
#include <stdio.h>

int fscanf( FILE* fp,
             const char* format,
             ... );
```

**Arguments:**
- `fp` The stream that you want to read from.
- `format` A string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

**Library:**
- `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `fscanf()` function scans input from the stream specified by `fp`, under control of the argument `format`.

**Returns:**
The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values (`errno` is set).

**Examples:**
Scan a date in the form “Friday March 26 1999”:
```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
```
```c

int day;
int year;
char weekday[10];
char month[10];
FILE *in_data;

in_data = fopen( "file", "r" );
if( in_data != NULL ) {
    fscanf( in_data, "%s %s %d %d",
            weekday, month, &day, &year );

    printf( "Weekday=%s Month=%s Day=%d Year=%d\n",
            weekday, month, day, year );

    fclose( in_data );
    return EXIT_SUCCESS;
}
return EXIT_FAILURE;

```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

errno, fwscanf(), scanf(), sscanf(), swscanf(), vscanf(), vfwscanf(),
vscanf(), vsscanf(), vswscanf(), vswscanf(), wscanf()
Synopsis:

```
#include <stdio.h>

int fseek( FILE* fp,  
            long offset,  
            int whence );

int fseeko( FILE* fp,  
            off_t offset,  
            int whence );
```

Arguments:

- `fp` A `FILE` pointer returned by `fopen()` or `freopen()`.
- `offset` The position to seek to, relative to one of the positions specified by `whence`.
- `whence` The position from which to apply the offset; one of:
  - `SEEK_SET` Compute the new file position relative to the start of the file. The value of `offset` must not be negative.
  - `SEEK_CUR` Compute the new file position relative to the current file position. The value of `offset` may be positive, negative or zero. A `SEEK_CUR` with a 0 `offset` is necessary when you want to switch from reading to writing on a stream opened for updates.
  - `SEEK_END` Compute the new file position relative to the end of the file.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `fseek()` function changes the current position of the stream specified by `fp`. This position defines the character that will be read or written by the next I/O operation on the file.

The `fseek()` function clears the end-of-file indicator, and undoes any effects of the `ungetc()` function on the stream.

You can use `ftell()` to get the current position of the stream before changing it. You can restore the position by using the value returned by `ftell()` in a subsequent call to `fseek()` with the `whence` parameter set to `SEEK_SET`.

Returns:

0 for success, or nonzero if an error occurs.

Errors:

If an error occurs, `errno` is set to indicate the type of error.

Examples:

Determine the size of a file, by saving and restoring the current position of the file:

```c
#include <stdio.h>
#include <stdlib.h>

long filesize( FILE *fp )
{
    long int save_pos;
    long size_of_file;

    /* Save the current position. */
    save_pos = ftell( fp );

    /* Jump to the end of the file. */
    fseek( fp, 0L, SEEK_END );

    /* Get the end position. */
    size_of_file = ftell( fp );

    /* Jump back to the original position. */

    return size_of_file;
}
```
fseek(), fseeko()

```c
fseek( fp, save_pos, SEEK_SET );
return( size_of_file );
}

int main( void )
{
    FILE *fp;
    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        printf( "File size=%ld\n", filesize( fp ) );
        fclose( fp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

- `fseek()` is ANSI, POSIX 1003.1; `fseeko()` is POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `errno`, `fgetpos()`, `fopen()`, `fsetpos()`, `ftell()`
Synopsis:

```c
#include <stdio.h>

int fsetpos( FILE* fp,
             const fpos_t* pos );
```

Arguments:

- `fp` The stream whose position you want to set.
- `pos` A pointer to a `fpos_t` object that specifies the new position for the stream. You must have initialized the value pointed to by `pos` by calling `fgetpos()` on the same file.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fsetpos()` function sets the current position of the stream specified by `fp` according to the value of the `fpos_t` object pointed to by `pos`.

Returns:

- `0` for success, or nonzero if an error occurs (`errno` is set).

Examples:

- See `fgetpos()`.

Classification:

- ANSI, POSIX 1003.1
### fsetpos()

#### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### See also:

errno, fgetpos(), fopen(), fseek(), ftell()
Synopsis:

```c
#include <sys/types.h>
#include <sys/stat.h>

int fstat( int filedes, 
           struct stat* buf );

int fstat64( int filedes, 
             struct stat64* buf );
```

Arguments:

- `filedes` The descriptor of the file that you want to get information about.
- `buf` A pointer to a buffer where the function can store the information about the file.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fstat()` and `fstat64()` functions get information from the file specified by `filedes` and stores it in the structure pointed to by `buf`.

The file `<sys/stat.h>` contains definitions for `struct stat`, as well as following macros:

- `S_ISBLK(m)` Test for block special file.
- `S_ISCHR(m)` Test for character special file.
- `S_ISDIR(m)` Test for directory.
- `S_ISFIFO(m)` Test for FIFO.
**fstat(), fstat64()**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>S_ISLNK(m)</code></td>
<td>Test for symbolic link.</td>
</tr>
<tr>
<td><code>S_ISREG(m)</code></td>
<td>Test for regular file.</td>
</tr>
<tr>
<td><code>S_TYPEISMQ(buf)</code></td>
<td>Test for message queue.</td>
</tr>
<tr>
<td><code>S_TYPEISSEM(buf)</code></td>
<td>Test for semaphore.</td>
</tr>
<tr>
<td><code>S_TYPEISSHM(buf)</code></td>
<td>Test for shared memory object.</td>
</tr>
</tbody>
</table>

The arguments to the macros are:

- `m`  The value of `st_mode` in a `stat` structure.
- `buf`  A pointer to a `stat` structure.

The macros evaluate to nonzero if the test is true, and zero if the test is false.

Access permissions are specified as a combination of bits in the `st_mode` field of the `stat` structure. These bits are defined in `<sys/stat.h>`. For more information, see “Access permissions” in the documentation for `stat()`.

The `st_mode` field also encodes the following bits:

- **S_ISUID**  Set user ID on execution. The process’s effective user ID (EUID) is set to that of the owner of the file when the file is run as a program. On a regular file, this bit may be cleared for security reasons on any write.
- **S_ISGID**  Set group ID on execution. Set effective group ID (EGID) on the process to the file’s group when the file is run as a program. On a regular file, this bit bit may be cleared for security reasons on any write.
fstat(), fstat64()

Returns:

0  Success.
-1  An error occurred (errno is set).

Errors:

EBADF        The filedes argument isn’t a valid file descriptor.
ENOSYS       The fstat() function isn’t implemented for the filesystem specified by filedes.
EOVERFLOW    The file size in bytes or the number of blocks allocated to the file or the file serial number can’t be represented correctly in the structure pointed to by buf.

Examples:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int filedes;
    int rc;
    struct stat buf;

    filedes = open( "file", O_RDONLY );
    if( filedes != -1 ) {
        rc = fstat( filedes, &buf );
        if( rc != -1 ) {
            printf( "File size = %d\n", buf.st_size );
        }
    }
    close( filedes );

    return EXIT_SUCCESS;
}

return EXIT_FAILURE;
```
Classification:

*fstat(*) is POSIX 1003.1; fstat64(*) is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

creat(*), dup(*), dup2(*), errno, fcntl(*), lstat(*), open(*), pipe(*), sopen(*), stat(*)

Synopsis:

```c
#include <sys/statvfs.h>

int fstatvfs( int fildes,
              struct statvfs *buf );

int fstatvfs64( int fildes,
                 struct statvfs64 *buf );
```

Arguments:

- `fildes` The descriptor for a file that resides on the filesystem that you want to get information about.
- `buf` A pointer to a buffer where the function can store information about the filesystem; see below.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fstatvfs()` function returns a “generic superblock” describing a filesystem; you can use it to get information about mounted filesystems. The `fstatvfs64()` function is a 64-bit version of `fstatvfs()`.

The `fildes` argument is an open file descriptor, obtained from a successful call to `open()`, `creat()`, `dup()`, `fcntl()`, or `pipe()`, for a file that resides on that filesystem. The filesystem type is known to the operating system. Read, write, or execute permission for the named file isn’t required.

The `buf` argument is a pointer to a `statvfs` or `statvfs64` structure that’s filled by the function. It contains at least:

- `unsigned long f_bsize`
  
  The preferred filesystem blocksize.
unsigned long \textit{f}_{frsize}  \\
   The fundamental filesystem blocksize (if supported)

\textit{fsblkcnt\_t f\_blocks}  \\
   The total number of blocks on the filesystem, in units of \textit{f}_{frsize}.

\textit{fsblkcnt\_t f\_bfree}  \\
   The total number of free blocks.

\textit{fsblkcnt\_t f\_bavail}  \\
   The number of free blocks available to a nonsuperuser.

\textit{fsfilcnt\_t f\_files}  \\
   The total number of file nodes (inodes).

\textit{fsfilcnt\_t f\_ffree}  \\
   The total number of free file nodes.

\textit{fsfilcnt\_t f\_favail}  \\
   The number of inodes available to a nonsuperuser.

unsigned long \textit{f}_{fsid}  \\
   The filesystem ID (dev for now).

char \textit{f\_basetype}[16]  \\
   The type of the target filesystem, as a null-terminated string.

unsigned long \textit{f\_flag}  \\
   A bitmask of flags; the function can set these flags:
   
   – \textit{ST\_RDONLY} — read-only filesystem.
   – \textit{ST\_NOSUID} — the filesystem doesn’t support \textit{setuid/setgid} semantics.

unsigned long \textit{f\_namemax}  \\
   The maximum filename length.
Returns:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (errno is set).</td>
</tr>
</tbody>
</table>

Errors:

- **EBADF** The `fildes` argument isn’t an open file descriptor.
- **EFAULT** The `buf` argument points to an illegal address.
- **EINTR** A signal was caught during execution.
- **EIO** An I/O error occurred while reading the filesystem.
- **EOVERFLOW** One of the values to be returned can’t be represented correctly in the structure pointed to by `buf`.

Classification:

`fstatvfs()` is POSIX 1003.1 XSI; `fstatvfs64()` is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The values returned for `f_files`, `f_ffree`, and `f_favail` might not be valid for NFS-mounted filesystems.
See also:

`chmod()`, `chown()`, `creat()`, `dup()`, `fcntl()`, `link()`, `mknod()`, `open()`,
`pipe()`, `read()`, `statfs()`, `statvfs64()`, `time()`, `unlink()`, `utime()`, `write()`
Synopsis:

```c
#include <unistd.h>

int fsync( int filedes );
```

Arguments:

`filedes`  The descriptor for the file that you want to synchronize.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fsync()` function forces all queued I/O operations for the file specified by the `filedes` file descriptor to finish, synchronizing the file’s state.

Although similar to `fdatasync()`, `fsync()` also guarantees the integrity of file information such as access and modification times.

Returns:

0 for success, or -1 if an error occurs (`errno` is set).

Errors:

- **EBADF**  The `filedes` argument isn’t a valid file descriptor open for writing.
- **EINVAL**  The implementation doesn’t support synchronized I/O for the given file.
- **ENOSYS**  The `fsync()` function isn’t supported for the filesystem specified by `filedes`. 
**fsync()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Classification:**

POSIX 1003.1 FSC

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`aio_fsync()`, `close()`, `fcntl()`, `fdatasync()`, `open()`, `read()`, `sync()`, `write()`
Synopsis:

```c
#include <stdio.h>

long int ftell( FILE* fp );

off_t ftello( FILE* fp );
```

Arguments:

`fp` The stream that you want to get the current position of.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ftell()` function returns the current position of the stream specified by `fp`. This position defines the character that will be read or written by the next I/O operation on the file. You can use the value returned by `ftell()` in a subsequent call to `fseek()` to restore the file position to a previous value.

The `ftello()` function is similar to `ftell()`, except that the position is returned as an `off_t`.

Returns:

The current position of the file or `-1L` if an error occurred (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

long filesize( FILE *fp )
{
    long int save_pos;
```
long size_of_file;

/* Save the current position. */
save_pos = ftell( fp );

/* Jump to the end of the file. */
seek( fp, 0L, SEEK_END );

/* Get the end position. */
size_of_file = ftell( fp );

/* Jump back to the original position. */
seek( fp, save_pos, SEEK_SET );

return( size_of_file );

int main( void )
{
    FILE *fp;
    fp = fopen( "file", "r" );

    if( fp != NULL ) {
        printf( "File size=%ld\n", filesize( fp ) );
        fclose( fp );

        return EXIT_SUCCESS;
    }

    return EXIT_FAILURE;
}

Classification:

ftell() is ANSI, POSIX 1003.1; ftello() is POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
See also:

`errno`, `fgetpos()`, `fopen()`, `fsetpos()`, `fseek()`
ftime()
Get the current time

Synopsis:
#include <sys/timeb.h>

int ftime( struct timeb * timeptr );

Arguments:
timeptr A pointer to a timeb structure where the function can store the current time; see below.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The ftime() function stores the current time in the timeptr structure. The timeb structure contains the following fields:

    time_t time  Time, in seconds, since the Unix Epoch, 00:00:00 January 1, 1970 Coordinated Universal Time (UTC).

    unsigned short millitm  Milliseconds.

    short timezone  Difference in minutes of the timezone from UTC.

    short dstflag  Nonzero if in daylight savings time.

Returns:
0  Success.
-1  An error occurred (errno is set).
Examples:

```c
#include <stdio.h>
#include <time.h>
#include <sys/timeb.h>
#include <stdlib.h>

int main( void )
{
    struct timeb timebuf;
    char *now;

    ftime( &timebuf );
    now = ctime( &timebuf.time );

    /* Note that we're cutting "now" off
     * after 19 characters to avoid the
     * \n that ctime() appends to the
     * formatted time string.
     */

    printf( "The time is %.19s.%hu\n",
            now, timebuf.millitm );

    return EXIT_SUCCESS;
}
```

Produces output similar to the following:

```
The time is Mon Jul 05 15:58:42.870
```

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
ftime()

See also:

asctime(), clock(), ctime(), difftime(), gmtime(), localtime(), mktime(), strftime(), time(), tzset()
ftruncate(), ftruncate64()

Truncate a file

Synopsis:

```c
#include <unistd.h>

int ftruncate( int fildes,
               off_t length );

int ftruncate64( int fildes,
                 off64_t length );
```

Arguments:

- `fildes` The descriptor for the file that you want to truncate.
- `length` The length that you want the file to be, in bytes.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These functions cause the file referenced by `fildes` to have a size of `length` bytes. If the size of the file previously exceeded `length`, the extra data is discarded (this is similar to using the F_FREESP option with `fcntl()`). If the size of the file was previously shorter than `length`, the file size is extended with NUL characters (similar to the F_ALLOCSP option to `fcntl()`).

The value of the seek pointer isn’t modified by a call to `ftruncate()`.

Upon successful completion, the `ftruncate()` function marks the `st_atime` and `st_mtime` fields of the file for update. If the `ftruncate()` function is unsuccessful, the file is unaffected.
ftruncate(), ftruncate64() © 2007, QNX Software Systems GmbH & Co. KG.

Returns:

Zero for success, or -1 if an error occurred (errno is set).

Errors:

EBADF The fildes argument isn’t a valid file descriptor.
EFBIG The file is a regular file and length is greater than the offset maximum associated with the file.
EINTR A signal was caught during the call to ftruncate().
EINVAL The fildes argument doesn’t refer to a file on which this operation is possible, the filedes argument isn’t open for writing or the length argument is less than the minimum file size for the specified filesystem.
EIO An I/O error occurred while reading from or writing to the filesystem.
ENOSYS The ftruncate() function isn’t implemented for the filesystem specified by filedes.
ENOTSUP The ftruncate() function is implemented for the specified filesystem, but the specific operation (F_ALLOCSP or F_FREESP; see fcntl()) isn’t supported.
EROFS The file resides on a read-only filesystem.

Classification:

ftruncate() is POSIX 1003.1; ftruncate64() is Large-file support

Safety

Cancellation point  No
Interrupt handler    No

continued...
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mmap(), open(), shm_open(), truncate()
**ftrylockfile()**

Acquire ownership of a file, without blocking

**Synopsis:**

```c
#include <stdio.h>

int ftrylockfile( FILE* file );
```

**Arguments:**

`file` A pointer to the `FILE` object for the file you want to lock.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ftrylockfile()` function is used by a thread to acquire ownership of a `FILE` if the object is available; `ftrylockfile()` is a nonblocking version of `flockfile()`.

**Returns:**

- **0** Success.
- **Nonzero** The lock can’t be acquired.

**Classification:**

POSIX 1003.1 TSF

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

© 2007, QNX Software Systems GmbH & Co. KG.
See also:

flockfile(), getc_unlocked(), getchar_unlocked(), putc_unlocked(), putchar_unlocked()
Walk a file tree

Synopsis:

```c
#include <ftw.h>

int ftw( const char *path,
         int (*fn)( const char *fname,
                    const struct stat *sbuf,
                    int flags),
         int depth );
```

Arguments:

- `path` The path of the directory whose file tree you want to walk.
- `fn` A pointer to a function that you want to call for each file; see below.
- `depth` The maximum number of file descriptors that `ftw()` can use. The `ftw()` function uses one file descriptor for each level in the tree.

  If `depth` is zero or negative, the effect is the same as if it were 1. The `depth` must not be greater than the number of file descriptors currently available for use. The `ftw()` function is faster if `depth` is at least as large as the number of levels in the tree.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

These functions are in `libc.a`, but not in `libc.so` (in order to save space).
**Description:**

The `ftw()` function recursively descends the directory hierarchy identified by `path`. For each object in the hierarchy, `ftw()` calls the user-defined function `fn()`, passing to it:

- a pointer to a NULL-terminated character string containing the name of the object
- a pointer to a `stat` structure (see `stat()`) containing information about the object
- an integer. Possible values of the integer, defined in the `<ftw.h>` header, are:

  - `FTW_F` The object is a file.
  - `FTW_D` The object is a directory.
  - `FTW_DNR` The object is a directory that can’t be read. Descendents of the directory aren’t processed.
  - `FTW_NS` The `stat()` failed on the object because the permissions weren’t appropriate, or the object is a symbolic link that points to a nonexistent file. The `stat` buffer passed to `fn()` is undefined.

The `ftw()` function visits a directory before visiting any of its descendents.

The tree traversal continues until the tree is exhausted, an invocation of `fn()` returns a nonzero value, or some error is detected within `ftw()` (such as an I/O error). If the tree is exhausted, `ftw()` returns zero. If `fn()` returns a nonzero value, `ftw()` stops its tree traversal and returns whatever value was returned by `fn()`.

When `ftw()` returns, it closes any file descriptors it opened; it doesn’t close any file descriptors that may have been opened by `fn()`.
Returns:

0  Success.
-1  An error (other than EACCESS) occurred (errno is set).

Classification:

ftw() is POSIX 1003.1 XSI  ftw64() is Large-file support

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Because ftw() is recursive, it might terminate with a memory fault when applied to very deep file structures.

This function uses malloc() to allocate dynamic storage during its operation. If ftw() is forcibly terminated, for example if longjmp() is executed by fn() or an interrupt routine, ftw() doesn’t have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have fn() return a nonzero value at its next invocation.

See also:

longjmp(), malloc(), nftw(), stat()
funlockfile()
Release ownership of a file

Synopsis:
#include <stdio.h>

void funlockfile( FILE* file );

Arguments:
file   A pointer to the FILE object for the file you want to unlock.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The funlockfile() function is used to release ownership of file granted to the thread. The behavior is undefined if a thread other than the current owner calls the funlockfile() function.

For more information, see flockfile().

Classification:
POSIX 1003.1 TSF

Safety
<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
funlockfile()

© 2007, QNX Software Systems GmbH & Co. KG.

See also:

flockfile(), ftrylockfile(), getc_unlocked(), getchar_unlocked(),
putc_unlocked(), putchar_unlocked()
Synopsis:

```c
#include <utime.h>

int futime( int fildes,
            const struct utimbuf *times );
```

```c
struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};
```

Arguments:

- `fildes`  The descriptor for the file whose modification time you want to get or set.
- `times`  NULL, or a pointer to a `utimbuf` structure where the function can store the modification time.

Library:

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `futime()` function records the modification time for the file or directory with the descriptor, `fildes`.

If the `times` argument is NULL, the access and modification times of the file or directory are set to the current time. The effective user ID of the process must match the owner of the file or directory, or the process must have write permission to the file or directory, or appropriate privileges in order to use the `futime()` function in this way.

If the `times` argument isn’t NULL, it’s interpreted as a pointer to a `utimbuf` structure, and the access and modification times of the file or directory are set to the values contained in the `actime` and `modtime` fields in this structure. Only the owner of the file or directory, and
processes with appropriate privileges are permitted to use the \futime()\ function in this way.

**Returns:**

0  Success.

-1  An error occurred (\errno\) is set.

**Errors:**

\EACCES\  Search permission is denied for a component of \path, or the \times\ argument is NULL, and the effective user ID of the process doesn’t match the owner of the file, and write access is denied.

\ENAMETOOLONG\  The argument \path\ exceeds \PATH_MAX\ in length, or a pathname component is longer than \NAME_MAX\.

\ENOENT\  The specified \path\ doesn’t exist, or \path\ is an empty string.

\ENOTDIR\  A component of \path\ isn’t a directory.

\EPERM\  The \times\ argument isn’t NULL, and the calling process’s effective user ID has write access to the file but doesn’t match the owner of the file, and the calling process doesn’t have the appropriate privileges.

\EROFS\  The named file resides on a read-only filesystem.

**Classification:**

Unix
futime()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, utime()
**fwide()**

Set or get the stream orientation

**Synopsis:**

```c
#include <wchar.h>

int fwide( FILE * fp,
          int mode );
```

**Arguments:**

- `fp`: The stream whose orientation you want to set.
- `mode`: The orientation mode:
  - If `mode` is greater than zero and the stream orientation hasn’t been set, `fwide()` flags the stream as wide-oriented.
  - If `mode` is less than zero, `fwide()` behaves similarly, but flags the stream as byte-oriented.
  - If `mode` is zero, `fwide()` returns the stream type without altering the stream.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `fwide()` function sets or determines the orientation of the stream `fp`.

**Returns:**

- `> 0`: The stream is (now) wide-oriented.
- `0`: The stream is unbound.
- `< 0`: The stream is (now) byte-oriented.
Errors:

EBADF The \textit{fp} argument isn’t valid.

Classification:

ANSI, POSIX 1003.1

Safety

\begin{tabular}{|l|}
\hline
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}

See also:

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
fwprintf()

Write wide-character output to a stream

Synopsis:

```c
#include <wchar.h>

int fwprintf( FILE * fp,
              const wchar_t * format,
              ... );
```

Arguments:

- `fp` The stream to which you want to send the output.
- `format` A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fwprintf()` function writes output to the stream specified by `fp`, under control of the `format` specifier.

The `fwprintf()` function is the wide-character version of `fprintf()`.

Returns:

The number of wide characters written, excluding the terminating `NUL`, or a negative number if an error occurred (`errno` is set).

Classification:

ANSI, POSIX 1003.1
fwprintf()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fprintf(), printf(), snprintf(), sprintf(), swprintf(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
fwrite()  © 2007, QNX Software Systems GmbH & Co. KG.
Write elements to a file

Synopsis:
#include <stdio.h>

size_t fwrite( const void* buf,
        size_t size,
        size_t num,
        FILE* fp );

Arguments:
buf  A pointer to a buffer that contains the elements that you want to write.
size The size of each element to write.
num  The number of elements to write.
fp   The stream to which to write the elements.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The fwrite() function writes num elements of size bytes each to the stream specified by fp.

Returns:
The number of complete elements successfully written; if an error occurs, this is less than num.

Errors:
If an error occurs, errno is set to indicate the type of error.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

struct student_data {
    int student_id;
    unsigned char marks[10];
};

int main( void )
{
    FILE *fp;
    struct student_data std;
    int i;

    fp = fopen( "file", "w" );
    if( fp != NULL ) {
        std.student_id = 1001;
        for( i = 0; i < 10; i++ ) {
            std.marks[i] = (unsigned char)(85 + i);
        }
        /* write student record with marks */
        i = fwrite( &std, sizeof( struct student_data ), 1, fp );
        printf( "Successfully wrote %d records\n", i );
        fclose( fp );

        if( i == 1 ) {
            return EXIT_SUCCESS;
        }
    }
    return EXIT_FAILURE;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

Cancellation point  Yes
Interrupt handler   No

continued...
fwrite()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, ferror(), fopen()
fwscanf()

Scan wide-character input from a stream

Synopsis:

```c
#include <wchar.h>

int fwscanf( FILE * fp,
             const wchar_t * format,
             ... );
```

Arguments:

- `fp` - The stream that you want to read from.
- `format` - A wide-character string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `fwscanf()` function scans input from the stream specified by `fp`, under control of the argument `format`. Following the format string is a list of addresses to receive values.

The `fwscanf()` function is the wide-character version of `fscanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.
fwscanf()
Return the string associated with a getaddrinfo() error code

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

const char * gai_strerror( int ecode );
```

Arguments:

- `ecode` The error code number from the `getaddrinfo()` function.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `gai_strerror()` function returns a string describing the error code from the `getaddrinfo()` function. Nonzero error codes are defined in `<netdb.h>` as follows:

- `EAI_ADDRFAMILY` The address family for `nodename` isn’t supported.
- `EAI_AGAIN` There was a temporary failure in name resolution.
- `EAI_BADFLAGS` Invalid value for `ai_flags`.
- `EAI_FAIL` Nonrecoverable failure in name resolution.
- `EAI_FAMILY` The `ai_family` isn’t supported.
- `EAI_MEMORY` Memory allocation failure.
- `EAI_NODATA` No address associated with the `nodename`.
- `EAI_NONAME` Either the `nodename` or the `servname` argument wasn’t provided or isn’t known.
**gai_strerror()**

EAI_SERVICE       The *servname* argument isn’t supported for
                  *ai_socktype*.
EAI_SOCKTYPE      The *ai_socktype* isn’t supported.
EAI_SYSTEM        System error returned in *errno*.

**Returns:**

If called with a proper *ecode* argument, a pointer to a string describing
the given error code. If the argument isn’t one of the EAI_* values, a
pointer to a string whose contents indicate an unknown error.

```
Don’t modify the strings that this function returns.
```

**Classification:**

POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`addrinfo`, `freeaddrinfo()`, `getaddrinfo()`
gamma(), gamma_r(),
gammaf(), gammaf_r()

Log gamma function

Synopsis:

```
#include <math.h>

double gamma( double x );
double gamma_r( double x, 
    int* signgam );

float gammaf( float x );
float gammaf_r( float x, 
    int* signgam );
```

Arguments:

- `x` An arbitrary number.
- `signgam` (gamma_r(), gammaf_r() only) A pointer to a location where the function can store the sign of \( \Gamma(x) \).

Library:

```
libm
```

Use the `-l m` option to `qcc` to link against this library.

Description:

The `gamma()` and `gamma_r()` functions return the natural log (\( \ln \)) of the `gamma()` function and are equivalent to `lgamma()`. These functions return \( \ln|\Gamma(x)| \), where \( \Gamma(x) \) is defined as follows:

\[
\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt
\]

For \( x > 0 \):

\[
\int_0^\infty e^{-t} t^{x-1} dt
\]

For \( x < 1 \): \( x / (\Gamma(1-x) * \sin(nx)) \)

The results converge when \( x \) is between 0 and 1. The \( \Gamma \) function has the property:
\[ \Gamma(N) = \Gamma(N-1) \times N \]

The \textit{gamma} functions compute the log because the \( \Gamma \) function grows very quickly.

The \textit{gamma()} and \textit{gammaf()} functions use the external integer \textit{signgam} to return the sign of \( \Gamma(x) \), while \textit{gamma_r()} and \textit{gammaf_r()} use the user-allocated space addressed by \textit{signgamp}.

The \textit{signgam} variable isn’t set until \textit{gamma()} or \textit{gammaf()} returns. For example, don’t use the expression:

\[
g = \text{signgam} \times \exp(\text{gamma}(x));
\]

to compute \( g = \Gamma(x)^\prime \). Instead, compute \textit{gamma()} first:

\[
lg = \text{gamma}(x);
g = \text{signgam} \times \exp(lg);
\]

Note that \( \Gamma(x) \) must overflow when \( x \) is large enough, underflow when \(-x\) is large enough, and generate a division by 0 exception at the singularities \( x \) a nonpositive integer.

**Returns:**

\[ \ln|\Gamma(x)| \]

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set \textit{errno} to 0, call the function, and then check \textit{errno} again. These functions don’t change \textit{errno} if no errors occurred.

**Classification:**

Legacy Unix
gamma(), gamma_r(),
gammaf(), gammaf_r()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

lgamma()
getaddrinfo()

Get socket address information

Synopsis:

```c
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo( const char * nodename,  
                 const char * servname,  
                 const struct addrinfo * hints,  
                 struct addrinfo ** res );
```

Arguments:

- **nodename** The node name. A non-NULL `nodename` may be either a node name or a numeric host address string (i.e. a dotted-decimal IPv4 address or an IPv6 hex address.)
- **servname** The server name. A non-NULL `servname` may be either a server name or a decimal port number.
- **hints** A pointer to an `addrinfo` structure that provides hints about the type of socket you’re supporting. See “Using the `hints` argument” for more information.
- **res** The address of a location where the function can store a pointer to a linked list of one or more `addrinfo` structures.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getaddrinfo()` function performs the functionality of `gethostbyname()` and `getservbyname()` but in a more sophisticated manner.

The `nodename` and `servname` arguments are either pointers to null-terminated strings or NULL. One or both of these two arguments
getaddrinfo() must be a non-NULL pointer. Normally, a client scenario specifies both nodename and servname.

On success, the getaddrinfo() function stores, in the location pointed to by res, a pointer to a linked list of one or more addrinfo structures. You can process each addrinfo structure in this list by following the ai_next pointer until reaching a NULL pointer. Each addrinfo structure contains the corresponding ai_family, ai_socktype, and ai_protocol arguments for a call to the socket() function. The ai_addr argument of the addrinfo structure points to a filled-in socket address structure with a length specified by the ai_addrlen argument.

Using the hints argument

You can optionally pass an addrinfo structure, pointed to by the hints argument, that provides hints concerning the type of socket that your application supports.

In this structure, all members — except ai_flags, ai_family, ai_socktype, and ai_protocol — must be zero or a NULL pointer. The addrinfo structure of the hints argument can accept various types of sockets:

<table>
<thead>
<tr>
<th>To accept:</th>
<th>Set:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any protocol family</td>
<td>ai_family</td>
<td>PF_UNSPEC</td>
</tr>
<tr>
<td>Any socket type</td>
<td>ai_socktype</td>
<td>0</td>
</tr>
<tr>
<td>Any protocol</td>
<td>ai_protocol</td>
<td>0</td>
</tr>
<tr>
<td>All of the above</td>
<td>hints</td>
<td>NULL</td>
</tr>
<tr>
<td>(as well as setting ai_flags to 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The hints argument defaults to all possibilities, but you can also use it to limit choices:

- If the application handles only TCP but not UDP, you could set the ai_socktype member of the hints structure to SOCK_STREAM.
getaddrinfo()

- If the application handles only IPv4 but not IPv6, you could set the `ai_family` member of the `hints` structure to PF_INET.

**Using the `ai_flags` argument in the `hints` structure**

You can set the `ai_flags` argument to further configure the `hints` structure. Settings for `ai_flags` include:

**AI_PASSIVE**
Set this bit if you plan to use the returned `addrinfo` structure in a call to `bind()`. In this call, if the `nodename` argument is a NULL pointer, then the IP address portion of the socket address structure `ai_addr` is set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address.

If you don’t set the `AI_PASSIVE` flag, you can use the returned `addrinfo` structure in a call to:

- `connect()` — connectionless or connection-oriented protocol
- `sendto()` — connectionless protocol
- `sendmsg()` — connectionless protocol

In this case, if the `nodename` argument is a NULL pointer, then the IP address portion of the socket address structure `ai_addr` is set to the loopback address.

**AI_CANONNAME**
Set this bit if you want the `ai_canonname` argument of the first `addrinfo` structure to point to a null-terminated string containing the canonical name of the specified `nodename`.

**AI_NUMERICHOST**
Set this bit if you want to prevent any type of name resolution service (such as DNS) from being used. A non-NULL `nodename` string must be a numeric
host address string; otherwise, `getaddrinfo()` returns `EAI_NONAME`.

**Pitfalls**

The arguments to `getaddrinfo()` must be sufficiently consistent and unambiguous or this function will return an error. Here are some problems you may encounter:

- Inconsistent hints — for Internet address families, specifying `SOCK_STREAM` for `ai_socktype` while specifying `IPPROTO_UDP` for `ai_protocol`.

- Inconsistent servname — specifying a servname that’s defined only for certain `ai_socktype` values, such as the TFTP service (a datagram service `SOCK_DGRAM`) on `SOCK_STREAM`.

- Undefined service names — specifying a servname while specifying `SOCK_RAW` for `ai_socktype`. (Service names aren’t defined for the internet `SOCK_RAW` space.)

- Incomplete specifications — specifying a numeric servname while leaving `ai_socktype` and `ai_protocol` unspecified. The `getaddrinfo()` function isn’t allowed to `glob()` the argument when a numeric servname doesn’t have a specified socket type.

The `getaddrinfo()` function dynamically allocates space for the following:

- `addrinfo` structures

- socket address structures

- canonical node name strings pointed to by the `addrinfo` structures.

Use `freeaddrinfo()` to free the `addrinfo` structures, and `gai_strerror()` to decipher error codes.
getaddrinfo()

Returns:

Zero for success, or nonzero if an error occurs.

Errors:

To get an explanation of any error code, use gai_strerror().

Examples:

The following code tries to connect to www.kame.net service HTTP using a stream socket. It loops through all the addresses available, regardless of the address family. If the destination resolves to an IPv4 address, it uses a AF_INET socket. Similarly, it uses an AF_INET6 socket if it resolves to IPv6. Note that there aren’t any hardcoded references to any particular address family; the code works even if getaddrinfo() returns addresses that aren’t IPv4/IPv6.

```c
struct addrinfo hints, *res, *res0;
int error;
int s;
const char *cause = NULL;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo("www.kame.net", "http", &hints, &res0);
if (error) {
    err1(1, "%s", gai_strerror(error));
    /*NOTREACHED*/
}
s = -1;
for (res = res0; res; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype,
               res->ai_protocol);
    if (s < 0) {
        cause = "socket";
        continue;
    }
    if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
        cause = "connect";
        close(s);
        s = -1;
        continue;
    }
}
```
The following example tries to open a wildcard-listening socket onto the HTTP service for all of the available address families:

```c
struct addrinfo hints, *res, *res0;
int error;
int s[MAXSOCK];
int nsock;
const char *cause = NULL;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
error = getaddrinfo(NULL, "http", &hints, &res0);
if (error) {
    err(1, "%s", gai_strerror(error));
    /*NOTREACHED*/
}
nsock = 0;
for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
    s[nsock] = socket(res->ai_family, res->ai_socktype,
                       res->ai_protocol);
    if (s[nsock] < 0) {
        cause = "socket";
        continue;
    }
    if (connect(s[nsock], res->ai_addr, res->ai_addrlen) < 0) {
        cause = "connect";
        close(s[nsock]);
        continue;
    }
    nsock++;
}
if (nsock == 0) {
    err(1, cause);
    /*NOTREACHED*/
}
freeaddrinfo(res0);
```
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

addrinfo, freeaddrinfo(), gai_strerror()
**Synopsis:**

```c
#include <stdio.h>

int getc( FILE* fp );
```

**Arguments:**

- `fp`  
  The stream you want to get the character from.

**Library:**

- `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `getc()` macro gets the next character from the stream designated by `fp`. The character is returned as an `int` value.

**Returns:**

The next character from the stream `fp`, cast as `(int)(unsigned char)`, or EOF if an end-of-file or error condition occurs (`errno` is set).

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE* fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
```
while((c = getc(fp)) != EOF) {
    putchar(c);
}
fclose(fp);
return EXIT_SUCCESS;
}
return EXIT_FAILURE;

Classification:
ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:
getc() is a macro.

See also:
errno, feof(), ferror(), fgetc(), fgetchar(), fgets(), fopen(), getchar(), gets(), putc(), putc_unlocked(), putchar(), putchar_unlocked(), ungetc()
getc_unlocked()

Get the next character from a file

Synopsis:

```c
#include <stdio.h>

int getc_unlocked( FILE *fp );
```

Arguments:

- `fp` The stream you want to get the character from.

Library:

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getc_unlocked()` function is a thread-unsafe version of `getc()`. You can use it safely only when the invoking thread has locked `fp` using `flockfile()` (or `ftrylockfile()`) and `funlockfile()`.

Returns:

The next character from the input stream pointed to by `fp`, or EOF if an end-of-file or error condition occurs (`errno` is set).

- Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

Classification:

- POSIX 1003.1 TSF

Safety

- Cancellation point: Yes

continued…
**getc_unlocked()**

**Safety**

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

feof(), ferror(), flockfile(), getc(), getchar(), getchar_unlocked(), putc(), putc_unlocked(), putchar(), putchar_unlocked()
Synopsis:

```
#include <stdio.h>

int getchar( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getchar()` function is equivalent to `getc()` on the `stdin` stream.

Returns:

The next character from the input stream pointed to by `stdin`, cast as `(int)(unsigned char)`, or EOF if an end-of-file or error condition occurs (`errno` is set).

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    /* Get characters from "file" instead of * stdin. */
    fp = freopen( "file", "r", stdin );
    while( ( c = getchar() ) != EOF ) {
        putchar(c);
    }
```
```c
fclose( fp );

return EXIT_SUCCESS;
```

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `feof()`, `ferror()`, `fgetc()`, `fgetchar()`, `getc()`, `putc()`, `putc_unlocked()`, `putchar()`, `putchar_unlocked()`
Synopsis:

```c
#include <stdio.h>

int getchar_unlocked( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getchar_unlocked()` function is a thread-unsafe version of `getchar()`. You can use it safely only when the invoking thread has locked `stdin` using `flockfile()` (or `ftrylockfile()`) and `funlockfile()`.

Returns:

The next character from the input stream pointed to by `stdin`, cast as `(int)(unsigned char)`, or EOF if an end-of-file or error condition occurs (`errno` is set).

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

Classification:

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued…
getchar_unlocked()

Safety

| Thread | No |

See also:

feof(), ferror(), getc(), getc_unlocked(), getchar(), putc(),
putc_unlocked(), putchar(), putchar_unlocked()
getcwd()

Get the name of the current working directory

Synopsis:

```c
#include <unistd.h>

char* getcwd( char* buffer,
             size_t size);
```

Arguments:

- **buffer**: A pointer to a buffer where the function can store the
directory name.
- **size**: The size of the buffer, in bytes.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getcwd()` function returns the name of the current working
directory. `buffer` is a pointer to a buffer of at least `size` bytes where the
NUL-terminated name of the current working directory will be placed.

The maximum size that might be required for `buffer` is PATH_MAX + 1 bytes. See `<limits.h>`.

Returns:

The address of the string containing the name of the current working
directory, or NULL if an error occurs (`errno` is set).

Errors:

- **EINVAL**: The argument `size` is negative or 0.
- **ELOOP**: Too many levels of symbolic links.
- **ENOSYS**: The `getcwd()` function isn’t implemented for the
  filesystem specified in the current working directory.
ERANGE The buffer is too small (as specified by size) to contain the name of the current working directory.

Examples:
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>

int main( void )
{
    char* cwd;
    char buff[PATH_MAX + 1];

    cwd = getcwd( buff, PATH_MAX + 1);
    if( cwd != NULL )
    {
        printf( "My working directory is %s.\n", cwd );
    }

    return EXIT_SUCCESS;
}
```
produces the output:

```
My working directory is /home/bill.
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
getcwd()

Caveats:

There is only one current working directory per process. In a multithreaded application, any thread calling chdir() will change the current working directory for all threads in that process.

See also:

chdir(), errno, mkdir(), rmdir()
**Synopsis:**

```c
#include <unistd.h>

int getdomainname( char * name,
                   size_t namelen );
```

**Arguments:**

- `name`: A buffer where the function can store the domain name.
- `namelen`: The size of the name array.

**Library:**

*libsocket*

Use the `-l socket` option to *qcc* to link against this library.

**Description:**

The `getdomainname()` function gets the standard domain name for the current processor and stores it in the buffer that `name` points to. The name is null-terminated.

If the buffer is too small, the name is truncated.

**Returns:**

- 0: Success.
- -1: An error occurred (`errno` is set).

**Errors:**

- `EFAULT`: The `name` or `namelen` parameters gave an invalid address.
Classification:

Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

setdomainname()
getdtablesize()  © 2007, QNX Software Systems GmbH & Co. KG.

Get the size of the file descriptor table

Synopsis:

```c
#include <unistd.h>

int getdtablesize( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

Each process has a fixed size descriptor table, which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The `getdtablesize()` returns the size of this table.

This function is equivalent to `getrlimit()` with the RLIMIT_NOFILE option.

Returns:

The size of the file descriptor table.

Classification:

Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`close()`, `dup()`, `getrlimit()`, `open()`, `select()`, `sysconf()`
getegid() — Get the effective group ID

Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>

gid_t getegid( void );
```

Library:

```
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `getegid()` function gets the effective group ID for the calling process.

Returns:

The calling process’s effective group ID. This function can’t fail.

Examples:

```c
/*
 * Print the effective group ID of a process
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "My effective group ID is %d\n", getegid() );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1
getegid()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getimeid(), getgid(), getuid(), setegid()
**getenv()**

Get the value of an environment variable

**Synopsis:**

```c
#include <stdlib.h>

char* getenv( const char* name );
```

**Arguments:**

- `name` The name of the environment variable whose value you want to get.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `getenv()` function searches the environment list for a string in the form `name=value` and returns a pointer to a string containing the value for the specified `name`. The matching is case-sensitive.

**Returns:**

- A pointer to the value assigned to `name`, or `NULL` if `name` wasn’t found in the environment.

  Don’t modify the returned string.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char* path;

    path = getenv( "INCLUDE" );
    if( path != NULL ) {
```
getenv()

```c
    printf("INCLUDE=%s\n", path);
    return EXIT_SUCCESS;
}

return EXIT_FAILURE;
```

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

The `getenv()` function manipulates the environment pointed to by the global `environ` variable.

**See also:**

`clearenv()`, `environ`, `execl()`, `execle()`, `execlpl()`, `execlepl()`, `execv()`,
`execve()`, `execvp()`, `execvpe()`, `putenv()`, `searchenv()`, `setenv()`,
`spawn*()` functions, `system()`
**Synopsis:**

```c
#include <sys/types.h>
#include <unistd.h>

uid_t geteuid( void );
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `geteuid()` function gets the effective user ID for the calling process.

**Returns:**

The calling process’s effective user ID.

**Examples:**

```c
/*
 * Print the effective user ID of a process.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "My effective user ID is %d\n", geteuid() );
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1
geteuid()

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getegid(), getgid(), getuid(), seteuid()
getgid()  
Get the group ID

Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>

gid_t getgid( void );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgid()` function gets the group ID for the calling process.

Returns:

The calling process’s group ID. This function can’t fail.

Examples:

```c
/*
 * Print the group id of a process.
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "I belong to group ID %d\n", getgid() );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1
### getgid()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`getegid()`, `geteuid()`, `getuid()`
Return an entry from the group database

Synopsis:

```
#include <grp.h>

struct group* getgrent(void);
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgrent()` function returns the next entry from the group database, although no particular order is guaranteed. This function uses a static buffer that’s overwritten by each call.

The `getgrent()`, `getgrgid()`, and `getgrnam()` function share the same static buffer.

Returns:

The next entry from the group database. When you first call `getgrent()`, the group database is opened. It remains open until either `getgrent()` returns NULL to signify end-of-file, or you call `endgrent()`.

Errors:

The `getgrent()` function uses the following functions, and as a result, `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
- `fopen()`
- `fseek()`
- `fseek()`
getgrent()

Examples:

/*
 * This program loops, reading a group name from
 * standard input and checking to see if it is a valid
 * group. If it isn’t valid, the entire contents of the
 * group database are printed.
 */
#include <stdio.h>
#include <stdlib.h>
#include <grp.h>
#include <limits.h>

int main( void )
{
    struct group* gr;
    char buf[80];

    setgrent();
    while( gets(buf) != NULL) {
        struct group* grn;
        if( (gr=getgrnam(buf)) != (struct group *)0) {
            printf("Valid group is: %s\n",gr->gr_name);
        } else {
            setgrent();
            while( (gr=getgrent()) != (struct group *)0 )
            printf("%s\n",gr->gr_name);
        }
    }
    endgrent();
    return( EXIT_SUCCESS );
}

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

endgrent(), errno, getgrgid(), getgrnam(), getpwent(), setgrent()
getgrgid()  
Get information about the group with a given ID

Synopsis:
```
#include <sys/types.h>
#include <grp.h>

struct group* getgrgid( gid_t gid );
```

Arguments:
- `gid`  
The ID of the group you want to get information about.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgrgid()` function lets a process gain more knowledge about group `gid`. This function uses a static buffer that’s overwritten by each call.

The `getgrent()`, `getgrgid()`, and `getgrnam()` functions share the same static buffer.

Returns:

A pointer to an object of type `struct group` containing an entry from the group database with a matching `gid`. On error or failure to find an entry with a matching `gid`, a NULL pointer is returned.

Examples:

```c
/*
 * Print a list of all users in your group
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```
#include <grp.h>

int main( void )
{
    struct group* g;
    char** p;

    if( ( g = getgrgid( getgid() ) ) == NULL ) {
        fprintf( stderr, "getgrgid: NULL pointer\n" );
        return( EXIT_FAILURE );
    }
    printf( "group name:%s\n", g->gr_name );
    for( p = g->gr_mem; *p != NULL; p++ ) {
        printf( "\t%s\n", *p );
    }
    return( EXIT_SUCCESS );
}

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

grent(), getgrgid_r(), getgrnam()
**Synopsis:**

```c
#include <sys/types.h>
#include <grp.h>

int getgrgid_r ( gid_t gid,
                struct group* grp,
                char* buffer,
                size_t bufsize,
                struct group** result );
```

**Arguments:**

- **gid**
  - The ID of the group you want to get information about.

- **grp**
  - A pointer to a `group` structure where the function can store information about the group.

- **buffer**
  - A buffer from which to allocate any memory required.

- **bufsize**
  - The size of the buffer.

- **result**
  - The address of a pointer that `getgrgid_r()` sets to the same pointer as `grp` on success, or to NULL if the function can’t find the group.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, `getgrgid_r()` updates the group structure pointed by `grp` and stores a pointer to that structure at the location pointed by `result`. The structure contains an entry from the group database with a matching `gid`.

This function allocates storage referenced by the group structure from the memory provided with the `buffer` parameter, which is `bufsize`
getgrijd_r()  © 2007, QNX Software Systems GmbH & Co. KG.

characters in size. You can determine the maximum size needed for this buffer by calling sysconf() with an argument of _SC_GETGR_R_SIZE_MAX.

The getgrijd_r() stores a NULL pointer at the location pointed by result on error or if the requested entry isn’t found.

Returns:

Zero for success, or an error number if an error occurred.

Errors:

ERANGE Insufficient storage was supplied via buffer and bufsize to contain the resulting group structure.

The getgrijd_r() function uses the following functions, and as a result, errno can be set to an error for any of these calls:

- fclose()
- fgets()
- fopen()
- fseek()
- fseek()
- rewind()

Classification:

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

852 C Library — F to H September 10, 2007
See also:

getgrgid(), getgrnam(), getgrnam_r(), getlogin(), sysconf()
getgrnam()

Get information about the group with a given name

Synopsis:

```c
#include <sys/types.h>
#include <grp.h>

struct group* getgrnam( const char* name );
```

Arguments:

- `name` The name of the group you want to get information about.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgrnam()` function lets a process gain more knowledge about the group named `name`. This function uses a static buffer that’s overwritten by each call.

The `getgrent()`, `getgrgid()`, and `getgrnam()` functions share the same static buffer.

Returns:

A pointer to an object of type `struct group` containing an entry from the group database with a matching name, or `NULL` on error or failure to find an entry with a matching name.

Examples:

```c
/*
 * Print the name of all users in the group given in argv[1]
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```
```c
#include <sys/types.h>
#include <grp.h>

int main( int argc, char** argv )
{
    struct group* g;
    char** p;

    if( ( g = getgrnam( argv[1] ) ) == NULL ) {
        fprintf( stderr, "getgrnam: %s failed
", argv[1] );
        return( EXIT_FAILURE );
    }
    printf( "group name:%s
", g->gr_name );
    for( p = g->gr_mem; *p != NULL; p++ ) {
        printf( "	%s
", *p );
    }
    return( EXIT_SUCCESS );
}
```

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`getgrent()`, `getgrgid()`, `getgrnam_r()`
getgrnam_r()

Get information about the group with a given name

Synopsis:

```c
#include <sys/types.h>
#include <grp.h>

int getgrnam_r( const char* name, 
    struct group* grp, 
    char* buffer, 
    size_t bufsize, 
    struct group** result );
```

Arguments:

- `name`: The name of the group you want to get information about.
- `grp`: A pointer to a group structure where the function can store information about the group.
- `buffer`: A buffer from which to allocate any memory required.
- `bufsize`: The size of the buffer.
- `result`: The address of a pointer that getgrgid_r() sets to the same pointer as `grp` on success, or to NULL if the function can’t find the group.

Library:

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

If _POSIX_THREAD_SAFE_FUNCTIONS is defined, the getgrnam_r() function updates the group structure pointed by `grp` and stores a pointer to that structure at the location pointed by `result`. The structure contains an entry from the group database with a matching name.

This function allocates storage referenced by the group structure from the memory provided with the `buffer` parameter, which is `bufsize`.
characters in size. You can determine the maximum size needed for this buffer by calling \texttt{sysconf()} with an argument of \_SC\_GETGR\_R\_SIZE\_MAX.

The \texttt{getgrnam\_r()} stores a NULL pointer at the location pointed by \texttt{result} on error or if the requested entry isn’t found.

**Returns:**

Zero for success, or an error number if an error occurred.

**Errors:**

ERANGE Insufficient storage was supplied via \texttt{buffer} and \texttt{bufsize} to contain the \texttt{group} structure.

The \texttt{getgrnam\_r()} function uses the following functions, and as a result, \texttt{errno} can be set to an error for any of these calls:

- \texttt{fclose()}
- \texttt{fgets()}
- \texttt{fopen()}
- \texttt{fseek()}
- \texttt{fseek()}
- \texttt{rewind()}

**Classification:**

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

getgrgid(), getgrgid_r(), getgrnam(), getlogin(), sysconf()
### Synopsis:

```c
#include <unistd.h>

int getgrouplist( const char *name, 
                  gid_t basegid, 
                  gid_t *groups, 
                  int *ngroups );
```

### Arguments:

- **name**
  - The name of the user.

- **basegid**
  - The basegid is automatically included in the list of groups. Typically this value is given as the group number from the password file.

  The Neutrino implementation of getgrouplist() ignores the basegid argument; see the “Caveats,” below.

- **groups**
  - A pointer to an array where the function can store the group IDs.

- **ngroups**
  - A pointer to the size of the groups array. The function sets the value pointed to by ngroups to be the actual number of groups found.

### Library:

- **libc**

  Use the -l c option to qcc to link against this library. This library is usually included automatically.

  This function is in libc.a, but not in libc.so (in order to save space).
**Description:**

The `getgrouplist()` function reads the group file and determines the group access list for the user specified in `name`.

**Returns:**

- **0** Success; the function fills in the group array and sets `*ngroups` to the number of groups found.
- **-1** The `groups` array is too small to hold all the user’s groups. The function fills the group array with as many groups as fit.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <limits.h>

int main()
{
    int ngroups, i;
    gid_t groups[NGROUPS_MAX];

    ngroups = NGROUPS_MAX;
    if (getgrouplist( getlogin(), getegid(), groups, &ngroups) == -1) {
        printf("Groups array is too small: %d\n", ngroups);
    }

    printf("%s belongs to these groups: %d", getlogin(), getegid());
    for (i=0; i < ngroups; i++) {
        printf("", %d", groups[i]);
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

**Files:**

- `/etc/group` Group membership list.
getgrouplist()

Classification:

Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

- The `getgrouplist()` function uses the routines based on `getgrent()`. If the invoking program uses any of these routines, the group structure will be overwritten in the call to `getgrouplist()`.

- This routine is BSD, and was designed for a system in which the effective group ID is placed in the supplementary group list. Neutrino doesn’t do this, so it ignores the `basegid` argument.

See also:

initgroups(), setgroups()
getgroups()

Get the supplementary group IDs of the calling process

Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>

int getgroups( int gidsetsize, 
    gid_t grouplist[] );
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gidsetsize</td>
<td>The size of the grouplist array.</td>
</tr>
<tr>
<td>grouplist</td>
<td>An array that the function can fill in with the process’s supplementary group IDs.</td>
</tr>
</tbody>
</table>

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getgroups()` function fills the array `grouplist` with the supplementary group IDs of the calling process. The values of array entries with indices greater than or equal to the returned value are undefined.

Returns:

The number of supplementary groups IDs; this value is zero if `NGROUPS_MAX` is zero. A value of -1 indicates an error (`errno` is set).

Errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The <code>gidsetsize</code> argument isn’t equal to zero, and is less than the number of supplementary group IDs.</td>
</tr>
</tbody>
</table>
getgroups()

Examples:

/*
 * Print the supplementary group IDs of
 * the calling process.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    int gidsize;
    gid_t *grouplist;
    int i;

    gidsize = getgroups( 0, NULL );
    grouplist = malloc( gidsize * sizeof( gid_t ) );
    getgroups( gidsize, grouplist );
    for( i = 0; i < gidsize; i++ )
        printf( "\n", ( int ) grouplist[i] );
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, getegid(), geteuid(), getgid(), getuid(), setgroups()
**gethostbyaddr()**

Get a network host entry, given an Internet address

**Synopsis:**

```c
#include <netdb.h>

struct hostent * gethostbyaddr( const void * addr, socklen_t len, int type );
```

**Arguments:**

- `addr` A pointer to the binary-format (i.e. not NULL-terminated) address in network byte order.
- `len` The length, in bytes, of `addr`.
- `type` The type of address. Currently, this must be AF_INET.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `gethostbyaddr()` function searches for information associated with a host, which has the address pointed to by `addr` within the address family specified by `type`, opening a connection to the database if necessary.

Both `gethostbyaddr()` and `gethostbyname()` are marked as obsolete in POSIX 1003.1. You should use `getaddrinfo()` or `getnameinfo()` instead.

This function returns a pointer to a structure of type `hostent` that describes an Internet host. This structure contains either the information obtained from a name server, or broken-out fields from a line in `/etc/hosts`.

You can use `sethostent()` to request the use of a connected TCP socket for queries. If the `stayopen` flag is nonzero, all queries to the name...
server will use TCP and the connection will be retained after each call to `gethostbyaddr()` or `gethostbyname()`. If the `stayopen` flag is zero, queries use UDP datagrams.

**Returns:**

A pointer to a valid `hostent` structure, or NULL if an error occurs (`h_errno` is set).

**Errors:**

See `herror()`.

**Examples:**

Use the `gethostbyaddr()` function to find a host:

```c
struct sockaddr_in client;
struct hostent* host;

int sock, fd, len;

...

len = sizeof( client );
fd = accept( sock, (struct sockaddr*)&client, &len );

if( fd == -1 ) { 
    perror( "accept" );
    exit( 1 );
}

host = gethostbyaddr( (const void*)&client.sin_addr, 
            sizeof(struct in_addr), 
            AF_INET );

printf( "Connection from %s: (%s)
", 
    host ? host->h_name : "<unknown>", 
    inet_ntoa( client.sin_addr ) );

...
```
gethostbyaddr()

Files:

/etc/hosts Host database file.
/etc/resolv.conf Resolver configuration file.

Classification:

POSIX 1003.1 OBS

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

This function uses static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it. Currently, only the Internet address format is understood.

See also:

endhostent(), gethostbyname(), gethostbyaddr_r(), gethostent(), herror(), hostent, sethostent()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent * gethostbyaddr_r(
    const void * addr,
    socklen_t length,
    int type,
    struct hostent * result,
    char * buffer,
    int buflen,
    int * h_errno);
```

Arguments:

- **addr**: A pointer to the binary-format (i.e. not NULL-terminated) address in network byte order.
- **length**: The length, in bytes, of `addr`.
- **type**: The type of address. Currently, this must be AF_INET.
- **result**: A pointer to a `struct hostent` where the function can store the host entry.
- **buffer**: A pointer to a buffer that the function can use during the operation to store host database entries; `buffer` should be large enough to hold all of the data associated with the host entry. A 2K buffer is usually more than enough; a 256-byte buffer is safe in most cases.
- **buflen**: The length of the area pointed to by `buffer`.
- **h_errno**: A pointer to a location where the function can store an `herrno` value if an error occurs.
gethostbyaddr_r()

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The gethostbyaddr_r() function is a thread-safe version of gethostbyaddr(). This function gets the network host entry for the host specified by addr. The addr argument is the network address of the specified network family, type. The buffer for addr is at least length bytes.

If you need to convert a text-based address into the format necessary for use as gethostbyaddr_r()’s addr, see inet_pton().

Returns:

A pointer to result, or NULL if an error occurs.

Errors:

If an error occurs, the int pointed to by h_errno is set to:

ERANGE The supplied buffer isn’t large enough to store the result.

HOST_NOT_FOUND Authoritative answer: Unknown host.

NO_ADDRESS No address associated with name; look for an MX record.

NO_DATA Valid name, but no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn’t a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).
gethostbyaddr_r()

NO_RECOVERY

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

TRY_AGAIN

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn’t receive a response from an authoritative server. A retry at some later time may succeed.

Files:

/etc/hosts Local host database file.

/etc/resolv.conf Resolver configuration file.

Classification:

Unix

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

gethostbyaddr(), gethostbyname(), gethostbyname_r(), inet_ntop(), inet_pton()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
**gethostbyname(), gethostbyname2()**

© 2007, QNX Software Systems GmbH & Co. KG.

*Get a network host entry, given a name*

**Synopsis:**

```c
#include <netdb.h>

struct hostent * gethostbyname( const char * name );

struct hostent * gethostbyname2( const char * name,
                                  int af );
```

**Arguments:**

- `name` The name of the Internet host whose entry you want to find.
- `af` *(gethostbyname2() only)* The address family; one of:
  - AF_INET
  - AF_INET6

**Library:**

*libsocket*

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `gethostbyname()` routine gets the network host entry for a given name. It returns a pointer to a structure of type `hostent` that describes an Internet host. This structure contains either the information obtained from a name server, or broken-out fields from a line in `/etc/hosts`.

Both `gethostbyaddr()` and `gethostbyname()` are marked as obsolete in POSIX 1003.1. You should use `getaddrinfo()` or `getnameinfo()` instead.

When using the name server, `gethostbyname()` searches for the named host in the current domain and in the domain’s parents, unless the name ends in a dot.
If the name doesn’t contain a dot, and the environment variable \texttt{HOSTALIASES} contains the name of an alias file, the alias file is first searched for an alias matching the input name. This file has the same form as /\texttt{etc}/\texttt{hosts}.

You can use \texttt{sethostent()} to request the use of a connected TCP socket for queries. If the \texttt{stayopen} flag is nonzero, all queries to the name server use TCP and the connection is retained after each call to \texttt{gethostbyname()} or \texttt{gethostbyaddr()}. If the \texttt{stayopen} flag is zero, queries use UDP datagrams.

The \texttt{gethostbyname2()} function is an evolution of the \texttt{gethostbyname()} function that lets you look up host names in address families other than AF_INET. If you specify an invalid address family, the function returns NULL and sets \texttt{h_errno} to NETDB\_INTERNAL.

**Returns:**

A pointer to a valid \texttt{hostent} structure, or NULL if an error occurs (\texttt{h_errno} is set).

**Errors:**

See \texttt{herror()}.

**Files:**

\texttt{/etc/hosts} Host database file.

\texttt{/etc/resolv.conf} Resolver configuration file.

For information about these files, see the \textit{Utilities Reference}.
Environment variables:

HOSTALIASES

Name of the alias file that gethostbyname() is to search first when the hostname doesn’t contain a dot.

Classification:

gethostbyname() is POSIX 1003.1 OBS; gethostbyname2() is QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

This function uses static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

See also:

endhostent(), gethostbyaddr(), gethostbyname_r(), gethostent(), herror(), hostent, sethostent()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
**gethostbyname_r()**

Get a network host entry by name

### Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent *gethostbyname_r(
    const char * name,
    struct hostent * result,
    char * buffer,
    int buflen,
    int * h_errno);
```

### Arguments:

- **name**
  The name of the Internet host whose entry you want to find.

- **result**
  A pointer to a `struct hostent` where the function can store the host entry.

- **buffer**
  A pointer to a buffer that the function can use during the operation to store host database entries; `buffer` should be large enough to hold all of the data associated with the host entry. A 2K buffer is usually more than enough; a 256-byte buffer is safe in most cases.

- ** buflen**
  The length of the area pointed to by `buffer`.

- **h_errno**
  A pointer to a location where the function can store an `h_errno` value if an error occurs.

### Library:

**libsocket**

Use the `-l socket` option to `qcc` to link against this library.
Description:

The `gethostbyname_r()` function is a thread-safe version of `gethostbyname()`. This function gets the network host entry for the host specified by `name`, and stores the entry in the `struct hostent` pointed to by `result`.

Returns:

A pointer to `result`, or NULL if an error occurs.

Errors:

If an error occurs, the `int` pointed to by `h_errno` is set to:

- **ERANGE** The supplied `buffer` isn’t large enough to store the result.
- **HOST_NOT_FOUND** Authoritative answer: Unknown host.
- **NO_ADDRESS** No address associated with name; look for an MX record.
- **NO_DATA** Valid name, but no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn’t a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).
- **NO_RECOVERY** Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.
- **TRY_AGAIN** Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn’t receive a response from
an authoritative server. A retry at some later time may succeed.

**Files:**

/etc/hosts  Local host database file.

/etc/resolv.conf  Resolver configuration file.

For information about these files, see the *Utilities Reference*.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`gethostbyaddr()`, `gethostbyaddr_r()`, `gethostname()`

`/etc/hosts`, `/etc/resolv.conf` in the *Utilities Reference*
**gethostent()**

*Read the next line of the host database file*

**Synopsis:**

```c
#include <netdb.h>

struct hostent * gethostent( void );
```

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `gethostent()` routine reads the next line in the host database file.

**Returns:**

A pointer to a valid `hostent` structure, or NULL if an error occurs.

**Files:**

- `/etc/hosts` Host database file.
- `/etc/resolv.conf` Resolver configuration file.

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
gethostent()

Caveats:

This function uses static data storage; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, this function understands only the Internet address format.

See also:

endhostent(), gethostbyaddr(), gethostbyname(), gethostent_r(), hostent, sethostent()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
gethostent_r()

Read the next line of the host database file

Synopsis:

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

struct hostent * gethostent_r( FILE ** hostf,  
    struct hostent * result,  
    char * buffer,  
    int buflen,  
    int * h_errno );

Arguments:

hostf NULL, or the address of the FILE * pointer associated with the host database file.

result A pointer to a struct hostent where the function can store the host entry.

buffer A pointer to a buffer that the function can use during the operation to store host database entries; buffer should be large enough to hold all of the data associated with the host entry. A 2K buffer is usually more than enough; a 256-byte buffer is safe in most cases.

buflen The length of the area pointed to by buffer.

h_errno A pointer to a location where the function can store an herrno value if an error occurs.

Library:

libsocket

Use the -l socket option to qcc to link against this library.
Description:

The `gethostent_r()` function is a thread-safe version of the `gethostent()` function. This function gets the local host’s entry. If the pointer pointed to by `hostf` is NULL, `gethostent_r()` opens `/etc/hosts` and returns its file pointer in `hostf` for later use. It’s the calling process’s responsibility to close the host file with `fclose()`.

The first time that you call `gethostent_r()`, pass NULL in the pointer pointed to by `hostf`.

Returns:

A pointer to `result`, or NULL if an error occurs.

Errors:

If an error occurs, the `int` pointed to by `h_errnop` is set to:

- **ERANGE**: The supplied `buffer` isn’t large enough to store the result.
- **HOST_NOT_FOUND**: Authoritative answer: Unknown host.
- **NO_ADDRESS**: No address associated with name, look for an MX record.
- **NO_DATA**: Valid name, no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn’t a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).
- **NO_RECOVERY**: Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.
TRY_AGAIN  Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn’t receive a response from an authoritative server. A retry at some later time may succeed.

Files:

/etc/hosts  Local host database file.

Classification:

Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

endhostent(), gethostent(), sethostent()

/etc/hosts in the Utilities Reference
gethostname()
Get the name of the current host

Synopsis:
#include <unistd.h>

    int gethostname( char * name,
                     size_t namelen );

Arguments:

    name            A buffer where the function can store the host name.
    namelen         The size of the buffer.

Library:

    libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The gethostname() function stores in name the standard hostname for
the current processor, as previously set by sethostname(). The
parameter namelen specifies the size of the name array. The returned
name is NULL-terminated unless insufficient space is provided.

    This function gets the value of the _CS_HOSTNAME configuration
    string, not that of the HOSTNAME environment variable.

Returns:

    0    Success.
    -1   An error occurred (errno isn’t set).
gethostname()

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Hostnames are limited to MAXHOSTNAMELEN characters (defined in <sys/param.h>).

See also:

sethostname()
getifaddrs()

Get a network interface address

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>

int getifaddrs( struct ifaddrs ** ifap );
```

Arguments:

- `ifap` The address of a location where the function can store a pointer to a linked list of `ifaddr` structures that contain the data related to the network interfaces on the local machine.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getifaddrs()` function stores a reference to a linked list of the network interfaces on the local machine in the memory referenced by `ifap`.

The data returned by `getifaddrs()` is dynamically allocated; you should free it by calling `freeifaddrs()` when you no longer need it.

Returns:

- `0` Success.
- `-1` An error occurred (`errno` is set).

Errors:

The `getifaddrs()` function may fail and set `errno` for any of the errors specified by:

- `ioctl()`
getifaddrs()

- malloc()
- socket()
- sysctl()

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, freeifaddrs(), ifaddrs, ioctl(), malloc(), socket(), sysctl()
GETIOVBASE()  
Get the base member of an iov_t structure

Synopsis:
#include <unistd.h>

#define GETIOVBASE(_iov) ...

Arguments:

_iov A pointer to the iov_t structure you want to get the base member from.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

This macro evaluates to the iov_base member of the given iov_t structure.

Returns:

The iov_base member of the iov_t structure, which is of type void *.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

GETIOVLEN(), SETIOV()
GETIOVLEN()
Get the length member of an iov_t structure

Synopsis:
```c
#include <unistd.h>

#define GETIOVLEN( _iov ) ...
```

Arguments:

`_iov` A pointer to the iov_t structure you want to get the length member from.

Library:

`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This macro evaluates to the `iov_len` member of the given iov_t structure.

Returns:

The `iov_len` member of the iov_t structure, which is of type `size_t`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

GETIOVBASE(), SETIOV()
getitimer()

Get the value of an interval timer

Synopsis:

```c
#include <sys/time.h>

int getitimer ( int which,
                struct itimerval *value );
```

Arguments:

- `which` The interval time whose value you want to get. Currently, this must be ITIMER_REAL.
- `value` A pointer to a `itimerval` structure where the function can store the value of the interval timer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The system provides each process with several interval timers, defined in `<sys/time.h>`. The `getitimer()` function stores the current value of the timer specified by `which` into the structure pointed to by `value`.

A timer value is defined by the `itimerval` structure (see `gettimeofday()` for the definition of `timeval`), which includes the following members:

```c
struct timeval it_interval; /* timer interval */
struct timeval it_value;   /* current value */
```

The `it_value` member indicates the time to the next timer expiration. The `it_interval` member specifies a value to be used in reloading `it_value` when the timer expires. Setting `it_value` to 0 disables a timer, regardless of the value of `it_interval`. Setting `it_interval` to 0 disables a timer after its next expiration (assuming `it_value` is nonzero).

Time values smaller than the resolution of the system clock are rounded up to the resolution of the system clock.
The interval timers include:

**ITIMER_REAL**

Decrements in real time. A SIGALRM signal is delivered when this timer expires.

**Returns:**

0  Success.

-1  An error occurred (**errno** is set).

**Errors:**

**EINVAL**  The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the **which** argument is unrecognized.

**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`alarm()`, `gettimeofday()`, `pthread_attr_setscope()`, `pthread_sigmash()`, `setitimer()`, `sigprocmask()`, `sleep()`, `sysconf()`
getlogin()

Get the user name associated with the calling process

Synopsis:

```c
#include <unistd.h>

char* getlogin( void ) ;
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getlogin()` function returns a pointer to a string containing the login name of the user associated with the calling process.

Returns:

A pointer to a string containing the user’s login name, or NULL if the user’s login name can’t be found.

The return value from `getlogin()` may point to static data and, therefore, may be overwritten by each call.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
See also:

getlogin_r(), getpwnam(), getpwuid()
getlogin_r()
Get the user name associated with the calling process

Synopsis:
#include <unistd.h>

int getlogin_r( char* name,
                size_t namesize );

Arguments:
name A buffer where the function can store the user name.
namesize The size of the buffer.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).

Description:
If _POSIX_THREAD_SAFE_FUNCTIONS is defined, the getlogin_r() function puts the login name of the user associated with the calling process in the character array pointed to by name. The array is namesize characters long and should have space for the name and the terminating NULL character. The maximum size of the login name is _POSIX_LOGIN_NAME_MAX.

If getlogin_r() is successful, name points to the name the user used at login, even if there are several login names with the same user ID.

Returns:
EOK Success.
ERANGE Insufficient storage was supplied via the name and namesize arguments to contain the user’s name.
getlogin_r()

Classification:

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

getlogin(), getpwnam(), getpwnam_r(), getpwuid(), getpwuid_r()
getnameinfo()

Perform address-to-nodename translation in a protocol-independent manner

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa,
                 socklen_t salen,
                 char *host, size_t hostlen,
                 char *serv, size_t servlen,
                 int flags);
```

Arguments:

- **sa**: Points to either a `sockaddr_in` structure (for IPv4) or a `sockaddr_in6` structure (for IPv6) that holds the IP address and port number.
- **salen**: Length of the `sockaddr_in` or `sockaddr_in6` structure.
- **host**: Buffer pointer for the host.
- **hostlen**: Size of the host buffer.
- **serv**: Buffer pointer for the server.
- **servlen**: Length of the server buffer.
- **flags**: Change the default action of `getnameinfo()`. By default, the fully qualified domain name (FQDN) for the host is looked up in the DNS and returned.

These flags are defined in `<netdb.h>`:

- **NI_NOFQDN**: Only the nodename portion of the FQDN is returned for local hosts.
- **NI_NUMERICHOST**: If set, or if the host’s name can’t be located in the DNS, the numeric form
of the host’s address is returned instead of its name (e.g. by calling inet_ntop() instead of getnodebyaddr()).

NI_NAMEREQD  If set, an error is returned when the host’s name can’t be located in the DNS.

NI_NUMERICSERV  If set, the numeric form of the service address (instead of its name) is returned e.g. its port number. You may require two NI_NUMERICxxx flags to support the -n flag that many commands provide.

NI_DGRAM  Specify that the service is a datagram service. Call getservbyport() with a second argument of udp instead of its default of tcp. This is required for the few ports (512-514) that have different services for UDP and TCP.

Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The getnameinfo() function defines and performs protocol-independent address-to-nodename translation. You can think of it as implementing the reverse-functionality of getaddrinfo() or similar functionality of gethostbyaddr() or getservbyport().

This function looks up an IP address and port number provided by the caller in the DNS and system-specific database. For both IP address and port number, the getnameinfo() function returns text strings in respective buffers provided by the caller. The function indicates successful completion by a zero return value; a non-zero return value indicates failure.
The `getnameinfo()` function returns the nodename associated with the IP address in the buffer pointed to by the `host` argument. The `hostlen` argument gives the length of this buffer.

The `getnameinfo()` function returns the service name associated with the port number in the buffer pointed to by the `serv` argument. The `servlen` argument gives the length of this buffer.

Specify zero for `hostlen` or `servlen` when the caller chooses not to return either string. Otherwise, the caller must provide buffers large enough to hold the nodename and the service name, including the terminating null characters.

Most systems don’t provide constants that specify the maximum size of either a FQDN or a service name. In order to aid your application in allocating buffers, the following constants are defined in `<netdb.h>`:

```
#define NI_MAXHOST 1025
#define NI_MAXSERV 32
```

You may find the first value as the constant MAXDNAME in recent versions of BIND’s `<arpa/nameser.h>`; older versions of BIND define this constant to be 256. The second value is a guess based on the services listed in the current Assigned Numbers RFC. BIND (Berkeley Internet Name Domain) is a suite of functionalities that implements Domain Name System (DNS) protocols.

**Extension**

The implementation allows experimental numeric IPv6 address notation with scope identifier. An IPv6 link-local address appears as string like `fe80::1%ne0`, when the NI_WITHSCOPEID bit is enabled in the `flags` argument. See `getaddrinfo()` for the notation.

**Returns:**

- **0** Success.
- Non-zero value
  
  An error occurred (see below).
Errors:

- **EAI_AGAIN**: The name couldn’t be resolved at this time. Future attempts may succeed.
- **EAI_BADFLAGS**: The flags had invalid values.
- **EAI_FAIL**: A nonrecoverable error occurred.
- **EAI_FAMILY**: The address family wasn’t recognized or the address length was invalid for the specified family.
- **EAI_MEMORY**: There was a memory allocation failure.
- **EAI_NONAME**: The name doesn’t resolve for the supplied parameters. NI_NAMEREQD is set and the host’s name can’t be located, or both node name and serv name were null.
- **EAI_SYSTEM**: A system error occurred. The error code can be found in errno.

Examples:

The following code gets the numeric hostname and the service name for a given socket address. There is no hardcoded reference to a particular address family.

```c
struct sockaddr *sa; /* input */
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
    errx(1, "could not get numeric hostname");
    /*NOTREACHED*/
}
printf("host=%s, serv=%s\n", hbuf, sbuf);
```

The following version checks if the socket address has reverse address mapping.

```c
struct sockaddr *sa; /* input */
char hbuf[NI_MAXHOST];
```
if (getnameinfo(sa, sa->sa_len, hbuf, sizeof(hbuf), NULL, 0, NI_NAMEREQD)) {
    errx(1, "could not resolve hostname"); /*NOTREACHED*/
}
printf("host=%s\n", hbuf);

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

getaddinfo(), gethostbyaddr(), getservbyport(), /etc/hosts, /etc/resolv.conf, /etc/services.named
getnetbyaddr()

Get a network entry, given an address (Unix)

Synopsis:

```
#include <netdb.h>

struct netent * getnetbyaddr( uint32_t net, int type );
```

Arguments:

- `net` The net address whose network entry you want to find.
- `type` The address type. This must currently be AF_INET.

Library:

```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getnetbyaddr()` function gets an entry for the given address, `net`, from the network database, `/etc/networks`.

This function returns a pointer to a structure of type `netent`, which contains the broken-out fields of a line in the network database.

The `setnetent()` function opens and rewinds the file. If you pass a nonzero `stayopen` argument to `setnetent()`, the network database isn’t closed after each call to `getnetbyname()`, or `getnetbyaddr()`.

The `getnetbyname()` and `getnetbyaddr()` functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

Returns:

A pointer to a valid `netent` structure, or NULL if an error occurs.
getnetbyaddr()

Files:

/etc/networks

Network name database file.

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Only Internet network numbers are currently understood.

See also:

endnetent(), getbyname(), getnetent(), netent, setnetent()

/etc/networks in the Utilities Reference
getnetbyname()  © 2007, QNX Software Systems GmbH & Co. KG.

Get a network entry, given a name

Synopsis:

```c
#include <netdb.h>

struct netent * getnetbyname( const char * name );
```

Arguments:

`name`  The name of the network whose entry you want to find.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getnetbyname()` function gets the network entry for the given name. This function returns a pointer to a structure of type `netent`, which contains the broken-out fields of a line in the network database, `/etc/networks`.

The `setnetent()` function opens and rewinds the file. If you pass a nonzero `stayopen` argument to `setnetent()`, the network database isn’t closed after each call to `getnetbyname()` or `getnetbyaddr()`.

The `getnetbyaddr()` and `getnetbyname()` functions sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

Returns:

A pointer to a valid `netent` structure, or NULL if an error occurs.

Files:

`/etc/networks`

Network name database file.

902  C Library — F to H  September 10, 2007
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

endnetent(), getnetbyaddr(), getnetent(), netent, setnetent()

/etc/networks in the Utilities Reference
Synopsis:

```c
#include <netdb.h>

struct netent * getnetent( void );
```

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getnetent()` function reads the next line of the network name database file, opening the file if necessary. It returns a pointer to a structure of type `netent`, which contains the broken-out fields of a line in the network database, `/etc/networks`.

Returns:

A pointer to a valid `netent` structure, or NULL if an error occurs.

Files:

`/etc/networks`

Network name database file.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

`endnetent()`, `getnetbyaddr()`, `getnetbyname()`, `netent`, `setnetent()`

`/etc/networks` in the Utilities Reference
Synopsis:

```c
#include <unistd.h>

int getopt( int argc,
            char * const argv[],
            const char * optstring );

extern char * optarg;
extern int optind, opterr, optopt;
```

Arguments:

- `argc` The argument count that was passed to `main()`.
- `argv` The argument array that was passed to `main()`.
- `optstring` A string of recognized option letters; if a letter is followed by a colon, the option takes an argument. Valid option characters for `optstring` consist of a single alphanumeric character (i.e. a letter or digit).

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getopt()` function is a command-line parser that can be used by applications that follow the Utility Syntax Guidelines described below.

The `optind` global variable is the index of the next element of the `argv[]` vector to be processed. The system initializes `optind` to 1 when the program is loaded, and `getopt()` updates it when it finishes with each element of `argv[]`. Reset `optind` to 1 if you want to use `getopt()` to process additional argument sets.
The `getopt()` function returns the next option character from `argv` that matches a letter in `optstring`, if there’s one that matches. If the option takes an argument, `getopt()` sets the global variable `optarg` to point to the option argument as follows:

1. If the option is the last letter in the string pointed to by an element of `argv`, then `optarg` contains the next element of `argv`, and `optind` is incremented by 2.
2. Otherwise, `optarg` points to the string following the option letter in that element of `argv`, and `optind` is incremented by 1.

The `getopt()` function returns -1 is returned and doesn’t change `optind` if:

- `argv[optind]` is NULL
- `argv[optind]` isn’t the character ‘-’
- `argv[optind]` points to the string "-".

This function returns -1 after incrementing `optind`, if:

- `argv[optind]` points to the string "--".

If `getopt()` encounters an option character that isn’t contained in `optstring`, it returns the `?` character. If it detects a missing option argument, it returns `:` if the first character of `optstring` is a colon; or `?` otherwise. In both cases, `getopt()` sets `optopt` to the option character that caused the error.

The `getopt()` always prints a diagnostic message to `stderr` unless `opterr` is set to 0, or the first character of `optstring` is a `:` character.

**Utility Syntax Guidelines**

The `getopt()` function may be used by applications that follow these guidelines:

- When describing the syntax of a utility, the options are listed in alphabetical order. There’s no implied relationship between the options based upon the order in which they appear, unless otherwise stated in the Options section, or:
- the options are documented as mutually-exclusive and such an option is documented to override any incompatible options preceding it
- when an option has option arguments repeated, the option and option argument combinations are interpreted in the order specified on the command line.

If an option that doesn’t have option arguments is repeated, the results depend on the application.

- Names of parameters that require substitution by actual values may be shown with embedded underscores or as `<parameter name>`. Angle brackets are used for the symbolic grouping of a phrase representing a single parameter and portable applications shouldn’t include them in data submitted to the utility.

- Options may be documented individually, or grouped (if they don’t take option arguments):

  ```
  utility_name [-a] [-b] [-c option_argument]
  [-d] [-e] [-f option_argument] [operand...]
  ```

  Or:

  ```
  utility_name [-ab] [-c option_argument]
  [-d] [-e] [-f option_argument] [operand...]
  ```

  Utilities with very complex arguments may be shown as:

  ```
  utility_name [options] [operand]
  ```

- Unless specified, whenever an operand or option argument is, or contains, a numeric value:

  - the number is interpreted as a decimal integer
  - numerals in the range 0 to 2,147,483,647 are syntactically recognized as numeric values
  - when the utility description states that it accepts negative numbers as operands or option arguments, numerals in the range -2,147,483,647 to 2,147,483,647 are syntactically recognized as numeric values
- ranges greater than those listed here are allowed.

All numbers within the allowable range aren’t necessarily semantically correct. A standard utility that accepts an option argument or operand that’s to be interpreted as a number, and for which a range of values smaller than that shown above is permitted, describes that smaller range along with the description of the option argument or operand. If an error is generated, the utility’s diagnostic message indicates that the value is out of the supported range, not that it’s syntactically incorrect.

- Arguments or option arguments enclosed in the “[“ and “]” notation are optional and can be omitted. Portable applications shouldn’t include the “[“ and “]” symbols in data submitted to the utility.

- Ellipses (...) are used to denote that one or more occurrences of an option or operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands may be specified. The forms:

  ```
  utility_name -f option_argument ... [operand ...]
  utility_name [-g option_argument] ... [operand ...]
  ```

  indicate that multiple occurrences of the option and its option argument preceding the ellipses are valid, with semantics as indicated in the Options section of the utility. In the first example, each option argument requires a preceding -f and at least one -f option_argument must be given.

- When the synopsis is too long to be printed on a single line in the documentation, the indented lines following the initial line are continuation lines. An actual use of the command appears on a single logical line.

**Returns:**

The next option character specified on the command line; a colon if a missing argument is detected and the first character of optstring is a colon; a question mark if an option character is encountered that’s not
in `optstring` and the first character of `optstring` isn’t a colon; otherwise, -1 when all command line options have been parsed.

**Examples:**

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int c, errflag = 0;

    while( ( c = getopt( argc, argv, "abt:" ) ) != -1 ) {
        switch( c ) {
            case 'a': printf( "apples\n" );
                break;
            case 'b': printf( "bananas\n" );
                break;
            case 't': printf( "tree = %s\n", optarg );
                break;
            case '?': ++errflag;
                break;
        }
    }
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

`getsubopt()`, `stderr`

getpass()  © 2007, QNX Software Systems GmbH & Co. KG.

Prompt for and read a password

Synopsis:

    #include <unistd.h>

    char *getpass( const char *prompt );

Arguments:

    prompt  The string you want to display to prompt for the password.

Library:

    libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).

Description:

The getpass() function can be used to get a password. It opens the current terminal, displays the given prompt, suppresses echoing, reads up to 32 characters into a static buffer, and restores echoing. This function adds a null character to the end of the string, but ignores additional characters and the newline character.

Returns:

    A pointer to the static buffer.

Classification:

    Legacy Unix
Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

This function leaves its result in an internal static buffer and returns a pointer to it. Subsequent calls to `getpass()` modify the same buffer. The calling process should zero the password as soon as possible to avoid leaving the clear-text password visible in the process’s address space.

See also:

`crypt()`
getpeername() © 2007, QNX Software Systems GmbH & Co. KG.
Get the name of the peer connected to a socket

Synopsis:
#include <sys/socket.h>

int getpeername( int s,
                struct sockaddr * name,
                socklen_t * namelen );

Arguments:

s The socket whose connected peer you want to get.

name A buffer where the function can store the name of the peer.

namelen A pointer to a socklen_t object that initially specifies the size of the buffer. This function stores the actual size of the name, in bytes, in this object.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The getpeername() function returns the name of the peer connected to socket s. The name is truncated if the buffer provided is too small.

Returns:

0 Success.

-1 An error occurred (errno is set).
getpeername()

Errors:

EBADF        Invalid descriptor s.
EFAULT       The name parameter points to memory not in a valid part of the process address space.
ENOBUSFS     Insufficient resources were available in the system to perform the operation.
ENOTCONN     The socket isn’t connected.

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

accept(), bind(), getsockname(), socket()
getpgid()  
Get a process group ID

Synopsis:

```c
#include <unistd.h>

pid_t getpgid( pid_t pid );
```

Arguments:

- `pid` The ID of the process whose process group ID you want to get.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getpgid()` returns the group ID for the process specified by `pid`. If `pid` is 0, `getpgid()` returns the calling process’s group ID.

The following definitions are worth mentioning:

- **Process**: An executing instance of a program, identified by a nonnegative integer called a process ID.
- **Process group**: A collection of one or more processes, with a unique process group ID. A process group ID is a positive integer.

Returns:

A process group ID for success, or (pid_t)-1 if an error occurs.

Errors:

If an error occurs, `errno` is set to:

- **ESRCH** The process specified by `pid` doesn’t exist.
getpgid()

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

getsid(), setpgid(), setsid()
getpgrp()  
Get the process group

Synopsis:
#include <sys/types.h>
#include <process.h>

pid_t getpgrp( void );

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The getpgrp() function gets the ID of the process group to which the calling process belongs.

Returns:
The calling process’s process group ID.

Examples:
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <sys/types.h>

int main( void )
{
    printf( "I am in process group %d\n", (int) getpgrp() );
    return EXIT_SUCCESS;
}

Classification:
POSIX 1003.1
getpgrp()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

setpgrp(), setsid()
getpid()  
Get the process ID

Synopsis:
```
#include <process.h>

pid_t getpid( void );
```

Library:
```
libc
```
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The `getpid()` function gets the process ID for the calling process.

Returns:
The process ID of the calling process.

Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>

int main ( void )
{
    printf( "I'm process %d\n", getpid() );
    return EXIT_SUCCESS;
}
```

Classification:
POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

continued...
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`getppid()`
getppid()
Get the parent process ID

Synopsis:

```
#include <sys/types.h>
#include <process.h>

pid_t getppid( void );
```

Library:

`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getppid()` function gets the process ID of the parent of the calling process.

Returns:

The calling process’s parent’s process ID.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <process.h>

int main( void )
{
    printf( "My parent is %d\n", getppid() );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1
getppid()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

getpid()
getprio()
Get the priority of a given process

Synopsis:

```c
#include <sched.h>

int getprio( pid_t pid );
```

Arguments:

- `pid` The process ID of the process whose priority you want to get.

Library:

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getprio()` function returns the current priority of thread 1 in process `pid`. If `pid` is zero, the priority of the calling thread is returned.

Returns:

The priority, or -1 if an error occurred (`errno` is set).

Errors:

- `ESRCH` The process `pid` doesn’t exist.

Classification:

- QNX 4

Safety

- Cancellation point: No
- Interrupt handler: No

continued...
**getprio()**

### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Caveats:

The `getprio()` and `setprio()` functions are included in the QNX Neutrino libraries for porting QNX 4 applications. For new programs, use `sched_getparam()` or `pthread_setschedparam()`.

### See also:

`errno`, `pthread_getschedparam()`, `pthread_setschedparam()`, `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getparam()`, `sched_setscheduler()`, `sched_setscheduler()`, `sched_yield()`, `setprio()`
getprotobynumber() © 2007, QNX Software Systems GmbH & Co. KG.
Get a protocol entry, given a name

Synopsis:

#include <netdb.h>

struct protoent * getprotobynumber( const char * name );

Arguments:

name The name of the protocol whose entry you want to get.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The getprotobynumber() function gets the entry for the given name from
the protocol database, /etc/protocols. This function returns a
pointer to a structure of type protoent, which contains the
broken-out fields of a line in the network protocol database.

The setprotoent() function opens and rewinds the file. If you pass a
nonzero stayopen argument to setprotoent(), the protocol database
isn’t closed after each call to getprotobynumber() or
getprotobynumber().

The getprotobynumber() and getprotobynumber() functions sequentially
search from the beginning of the file until a matching protocol name
or protocol number is found, or until EOF is encountered.

Returns:

A pointer to a valid protoent structure, or NULL if an error occurs.

Files:

/etc/protocols

Protocol name database file.
getprotobyname()

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

See also:

endprotoent(), getprotobynumber(), getprotoent(), protoent, setprotoent()

/etc/protocols in the Utilities Reference
Synopsis:

```c
#include <netdb.h>

struct protoent * getprotobynumber( int proto );
```

Arguments:

- `proto` The protocol number whose entry you want to get.

Library:

- `libsocket`

  Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getprotobynumber()` function gets the protocol entry for the given number. It returns a pointer to structure of type `protoent`, which contains the broken-out fields of a line in the network protocol database, `/etc/protocols`.

The `setprotoent()` function opens and rewinds the file. If you pass a nonzero `stayopen` argument to `setprotoent()`, the protocol database isn’t closed after each call to `getprotobyname()` or `getprotobynumber()`.

The `getprotobyname()` and `getprotobynumber()` functions sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

Returns:

A pointer to a valid `protoent` structure, or NULL if an error occurs.

Files:

- `/etc/protocols`

  Protocol name database file.
getprotobynumber()

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

**See also:**

`endprotoent()`, `getprotobyname()`, `getprotoent()`, `protoent()`, `setprotoent()`

`/etc/protocols` in the *Utilities Reference*
Synopsis:

```c
#include <netdb.h>

struct protoent * getprotoent( void );
```

Library:

```c
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getprotoent()` function reads the next line of the protocol name database file, opening the file if necessary. It returns a pointer to a structure of type `protoent`, which contains the broken-out fields of a line in the network protocol database, `/etc/protocols`.

Returns:

A pointer to a valid `protoent` structure, or `NULL` if an error occurs.

Files:

```
/etc/protocols
```

Protocol name database file.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

See also:

endprotoent(), getprotobynumber(), getprotobynumber(), protoent, setprotoent()

/etc/protocols in the Utilities Reference
**getpwent()**

Get an entry from the password database

**Synopsis:**

```c
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwent( void );
```

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `getpwent()` function returns the next entry from the password database. This function uses a static buffer that’s overwritten by each call.

The `getpwent()`, `getpwnam()`, and `getpwuid()`, functions share the same static buffer.

**Returns:**

A pointer to an object of type `struct passwd` containing the next entry from the password database. When `getpwent()` is first called, the password database is opened, and remains open until either a NULL is returned to signify end-of-file, or `endpwent()` is called.

**Errors:**

The `getpwent()` function uses the following functions, and as a result, `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
- `fopen()`
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main( void )
{
    struct passwd* pw;
    char   buf[80];

    setpwent( );
    while( gets( buf ) != NULL ) {
        if( ( pw = getpwnam( buf ) ) != ( struct passwd * )0 ) {
            printf( "Valid login name is: %s\n", pw->pw_name );
        } else {
            setpwent( );
            while( ( pw=getpwent( ) ) != ( struct passwd * )0 )
                printf( "%s\n", pw->pw_name );
        }
    }
    endpwent();
    return( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1 XSI

Safety

Cancellation point  Yes
Interrupt handler  No

continued...
getpwent()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

endpwent(), errno, getgrent(), getlogin(), getpwnam(), getpwuid(), setpwent()
getpwnam()

Get information about the user with a given name

Synopsis:

```c
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwnam( const char* name );
```

Arguments:

- **name**: The name of the user whose entry you want to find.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getpwnam()` function gets information about the user with the given `name`. It uses a static buffer that’s overwritten by each call.

```
The getpwent(), getpwnam(), and getpwuid() functions share the same static buffer.
```

The `getpwnam_r()` function is a reentrant version of `getpwnam()`.

Returns:

A pointer to an object of type `struct passwd` containing an entry from the group database with a matching `name`. A NULL pointer is returned on error or failure to find a entry with a matching `name`.

Examples:

```c
/*
 * Print information from the password entry
 * about the user name given as argv[1].
 */
#include <stdio.h>
#include <unistd.h>
```
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>

int main( int argc, char* *argv )
{
    struct passwd* pw;

    if( ( pw = getpwnam( argv[1] ) ) == NULL ) {
        fprintf( stderr, "getpwnam: unknown %s
", argv[1] );
        return( EXIT_FAILURE );
    }
    printf( "login name %s
", pw->pw_name );
    printf( "user id %d
", pw->pw_uid );
    printf( "group id %d
", pw->pw_gid );
    printf( "home dir %s
", pw->pw_dir );
    printf( "login shell %s
", pw->pw_shell );
    return( EXIT_SUCCESS );
}

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

getlogin(), getpwent(), getpwnam_r(), getpwuid()
getpwnam_r()

Get information about the user with a given name

Synopsis:

```c
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r( const char* name,
                 struct passwd* pwd,
                 char* buffer,
                 size_t bufsize,
                 struct passwd* result );
```

Arguments:

- **name**: The name of the user whose entry you want to find.
- **pwd**: A pointer to a `passwd` structure where the function can store the entry.
- **buffer**: A block of memory that the function can use to allocate storage referenced by the `passwd` structure. You can determine the maximum size needed for this buffer by calling `sysconf()` with an argument of `_SC_GETPW_R_SIZE_MAX`.
- **bufsize**: The size of the block that `buffer` points to, in characters.
- **result**: The address of a pointer to a `passwd` structure. If `getpwnam_r()` finds the entry, it stores a pointer to `pwd` in the location indicated by `result`; otherwise the function stores a NULL pointer there.

Library:

- **libc**

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.
Description:
The `getpwnam_r()` function is a reentrant version of `getpwnam()`. It gets information about the user with the given name.

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, the `getpwnam_r()` function updates the `passwd` structure pointed to by `pwd` and stores a pointer to that structure at the location pointed by `result`. The structure contains an entry from the user database with the given name.

The function stores a NULL pointer at the location pointed by `result` on error or if it can’t find the requested entry.

Returns:
Zero for success, or an error number.

Errors:

ERANGE Insufficient storage was supplied via `buffer` and `bufsize` to contain the resulting `passwd` structure.

The `getpwnam_r()` function uses the following functions, and as a result, `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
- `fopen()`
- `fseek()`
- `fseek()`
- `rewind()`

Classification:
POSIX 1003.1 TSF
getpwnam_r()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getlogin(), getpwent(), getpwnam(), getpwuid(), getpwuid_r()
getpwuid() Get information about the user with a given ID

Synopsis:

```c
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwuid( uid_t uid );
```

Arguments:

`uid` The userid whose entry you want to find.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getpwuid()` function gets information about user `uid`. This function uses a static buffer that’s overwritten by each call.

The `getpwent()`, `getpwnam()`, and `getpwuid()` functions share the same static buffer.

Returns:

A pointer to an object of type `struct passwd` containing an entry from the group database with a matching `uid`, or NULL if an error occurred or the function couldn’t find a matching entry.

Examples:

```c
/*
 * Print password info on the current user.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pwd.h>
```
```c
int main( void )
{
    struct passwd* pw;

    if( ( pw = getpwuid( getuid() ) ) == NULL ) {
        fprintf( stderr,
                "getpwuid: no password entry\n" );
        return( EXIT_FAILURE );
    }
    printf( "login name %s\n", pw->pw_name );
    printf( "user id %d\n", pw->pw_uid );
    printf( "group id %d\n", pw->pw_gid );
    printf( "home dir %s\n", pw->pw_dir );
    printf( "login shell %s\n", pw->pw_shell );
    return( EXIT_SUCCESS );
}
```

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`getlogin()`, `getpwent()`, `getpwnam()`
getpwuid_r()

Get information about the user with a given ID

Synopsis:

```c
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r( uid_t uid,
    struct passwd* pwd,
    char* buffer,
    size_t bufsize,
    struct passwd** result );
```

Arguments:

- **uid**  
The userid whose entry you want to find.
- **pwd**  
A pointer to a `passwd` structure where the function can store the entry.
- **buffer**  
A block of memory that the function can use to allocate storage referenced by the `passwd` structure. You can determine the maximum size needed for this buffer by calling `sysconf()` with an argument of 
  `_SC_GETPW_R_SIZE_MAX`.
- **bufsize**  
The size of the block that `buffer` points to, in characters.
- **result**  
The address of a pointer to a `passwd` structure. If `getpwnam_r()` finds the entry, it stores a pointer to `pwd` in the location indicated by `result`; otherwise the function stores a NULL pointer there.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:
The `getpwuid_r()` function is a reentrant version of `getpwuid()`. It lets a process gain more knowledge about user with the given uid.

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, the `getpwuid_r()` function updates the `passwd` structure pointed to by `pwd` and stores a pointer to that structure at the location pointed by `result`. The structure contains an entry from the user database with a matching uid.

The function stores a NULL pointer at the location pointed by `result` on error or if it can’t find the requested entry.

Returns:
Zero for success, or an error number.

Errors:
ERANGE Insufficient storage was supplied via `buffer` and `bufsize` to contain the resulting `passwd` structure.

The `getpwuid_r()` function uses the following functions, and as a result, `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
- `fopen()`
- `fseek()`
- `rewind()`

Classification:
POSIX 1003.1 TSF
**getpwuid_r()**

© 2007, QNX Software Systems GmbH & Co. KG.

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

getlogin(), getpwnam(), getpwnam_r(), getpwuid()
**getrlimit()**, **getrlimit64()**

Get the limit on a system resource

**Synopsis:**

```c
#include <sys/resource.h>

int getrlimit( int resource,
               struct rlimit * rlp);

int getrlimit64( int resource,
                 struct rlimit64 * rlp);
```

**Arguments:**

- **resource**
  The resource whose limit you want to get; one of the following:
  - RLIMIT_AS
  - RLIMIT_CORE
  - RLIMIT_CPU
  - RLIMIT_DATA
  - RLIMIT_FSIZE
  - RLIMIT_MEMLOCK
  - RLIMIT_NOFILE
  - RLIMIT_NPROC
  - RLIMIT_NTHR
  - RLIMIT_OFILE
  - RLIMIT_RSS
  - RLIMIT_STACK
  - RLIMIT_VMEM

  For descriptions and the actions taken when the current limit is exceeded, see `setrlimit()`.

- **rlp**
  A pointer to a `rlimit` or `rlimit64` structure where the function can store the limit on the resource. The `rlimit` and `rlimit64` structures include at least the following members:
getrlimit(), getrlimit64() © 2007, QNX Software Systems GmbH & Co. KG.

```c
rlim_t rlim_cur; /* current (soft) limit */
rlim_t rlim_max; /* hard limit */
```

The `rlim_t` type is an arithmetic data type to which you can cast objects of type `int`, `size_t`, and `off_t` without loss of information.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getrlimit()` function gets the limits on the consumption of a variety of system resources by a process and each process it creates. The `getrlimit64()` function is a 64-bit version of `getrlimit()`.

Each call to `getrlimit()` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values:

- the current (soft) limit
- a maximum (hard) limit.

A process can change soft limits to any value that’s less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that’s greater than or equal to the soft limit. Only a process with an effective user ID of `root` can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit()` subject to the constraints described above. Limits may have an “infinite” value of `RLIM_INFINITY`.

Because limit information is stored in the per-process information, the shell builtin `ulimit` command (see the entry for `ksh` in the `Utilities Reference`) must directly execute this system call if it’s to affect all future processes created by the shell.

The values of the current limit of the following resources affect these parameters:
### Resource Parameters

<table>
<thead>
<tr>
<th>Resource</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_FSIZE</td>
<td>FCHR_MAX</td>
</tr>
<tr>
<td>RLIMIT_NOFILE</td>
<td>OPEN_MAX</td>
</tr>
</tbody>
</table>

When using `getrlimit()`, if a resource limit can be represented correctly in an object of type `rlim_t`, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned is `RLIM_SAVED_MAX`; otherwise, the value returned is `RLIM_SAVED_CUR`.

A limit whose value is greater than `RLIM_INFINITY` is permitted.

The `exec*` family of functions also causes resource limits to be saved.

### Returns:

- **0**  
  Success.

- **-1**  
  An error occurred (`errno` is set).

### Errors:

- **EFAULT**  
  The `rlp` argument points to an illegal address.

- **EINVAL**  
  An invalid resource was specified.

- **EPERM**  
  The limit specified to `setrlimit()` would have raised the maximum limit value, and the effective user of the calling process isn’t the superuser.

### Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

int main( void )
{
    struct rlimit curr_limits;
```
if (getrlimit (RLIMIT_NPROC, &curr_limits) == -1) {
    perror ("The call to getrlimit() failed.");
    return EXIT_FAILURE;
} else {
    printf ("The current maximum number of processes is %d.\n",
            (int) curr_limits.rlim_cur);
    printf ("The hard limit on the number of processes is %d.\n",
            (int) curr_limits.rlim_max);
}
return EXIT_SUCCESS;

Classification:

getrlimit() is POSIX 1003.1 XSI; getrlimit64() is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

brk(), execl(), execle(), execvp(), execvep(), execv(), execve(), execvp(),
execvep(), fork(), getdtablesize(), malloc(), open(), setrlimit(),
setrlimit64(), signal(), sysconf()

ulimit builtin command (see the entry for ksh in the Utilities Reference)
getrusage()

Get information about resource utilization

Synopsis:

```c
#include <sys/resource.h>

int getrusage( int who,
               struct rusage * r_usage );
```

Arguments:

`who` Which process to get the usage for:

- `RUSAGE_CHILDREN` — get information about resources used by the terminated and waited-for children of the current process. If the child is never waited for (e.g. if the parent has SA_NOCLDWAIT set, or sets SIGCHLD to SIG_IGN), the resource information for the child process is discarded and isn’t included.
- `RUSAGE_SELF` — get information about resources used by the current process.

`r_usage` A pointer to an object of type `struct rusage` in which the function can store the resource information; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getrusage()` function provides measures of the resources used by the current process or its terminated and waited-for child processes, depending on the value of the `who` argument.

The `rusage` structure is defined as:

```c
struct timeval ru_utime; /* user time used */
```
getusage()

The members include:

- **ru_utime**: The total amount of time, in seconds and microseconds, spent executing in user mode.
- **ru_stime**: The total amount of time, in seconds and microseconds, spent executing in system mode.
- **ru_maxrss**: The maximum resident set size, given in pages. See the Caveats section, below.
- **ru_ixrss**: Not currently supported.
- **ru_idrss**: An “integral” value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of the process running when a clock tick occurs. The value is given in pages times clock ticks. It doesn’t take sharing into account. See the Caveats section, below.
- **ru_isrss**: Not currently supported.
- **ru_minflt**: The number of page faults serviced that didn’t require any physical I/O activity. See the Caveats section, below.
getrusage()

```
ru_majflt The number of page faults serviced that required
physical I/O activity. This could include page ahead
operations by the kernel. See the Caveats section,
below
ru_nswap The number of times a process was swapped out of
main memory.
ru_inblock The number of times the file system had to perform
input in servicing a read() request.
ru_oublock The number of times the filesystem had to perform
output in servicing a write() request.
ru_msgsnd The number of messages sent over sockets.
ru_msgrecv The number of messages received from sockets.
ru_nsignals The number of signals delivered.
ru_nvcsw The number of times a context switch resulted due
to a process’s voluntarily giving up the processor
before its timeslice was completed (usually to await
availability of a resource).
ru_nivcsw The number of times a context switch resulted due
to a higher priority process’s becoming runnable or
because the current process exceeded its time slice.
```

**Returns:**

0 Success.

-1 An error occurred (errno is set).

**Errors:**

EFAULT The address specified by the r_usage argument isn’t in a
valid portion of the process’s address space.

EINVAL Invalid who parameter.
getusage()

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Only the timeval fields of struct rusage are supported.

The numbers ru_inblock and ru_oublock account only for real I/O, and are approximate measures at best. Data supplied by the cache mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.

Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page that isn’t in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing read() and write() functions. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware-detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, major page faults require physical I/O, while minor page faults don’t. For example, reclaiming the page from the free list...
would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in memory. For example, if an address space faults on some “hot” executable or shared library, a minor page fault results for the address space. Also, anyone doing a `read()` or `write()` to something that’s in the page cache gets a minor page fault(s) as well.

There’s no way to obtain information about a child process that hasn’t yet terminated.

See also:

`gettimeofday()`, `read()`, `times()`, `wait()`, `write()`
get\

Synopsis:

```c
#include <stdio.h>

char *gets( char *buf);
```

Arguments:

- `buf`: A buffer where the function can store the string.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `gets()` function gets a string of characters from the `stdin` stream, and stores them in the array pointed to by `buf` until end-of-file is encountered or a newline character is read. Any newline character is discarded, and the string is NUL-terminated.

You should use `fgets()` instead of `gets()`: `gets()` happily overflows the `buf` array if a newline character isn’t read from `stdin` before the end of the array is reached.

The `gets()` function is similar to `fgets()`, except that `gets()` operates with `stdin`, has no size argument, and replaces a newline character with the NUL character.

Returns:

A pointer to `buf`, or NULL when end-of-file is encountered before reading any characters or a read error occurred (`errno` is set).
Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

Examples:
```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];

    while( gets( buffer ) != NULL ) {
        puts( buffer );
    }

    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

See also:

- `errno`
- `feof()`
- `ferror()`
- `fopen()`
- `getc()`
- `fgetc()`
- `fgets()`
- `puts()`
- `ungetc()`
getservbyname()

Get a service entry, given a name

Synopsis:

```c
#include <netdb.h>

struct servent * getservbyname( const char * name, const char * proto );
```

Arguments:

- `name` The name of the service whose entry you want to find.
- `proto` NULL, or the protocol for the service.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getservbyname()` function gets the entry for the given name and protocol from the network services database, `/etc/services`. This function returns a pointer of type `servent`, which contains the broken-out fields of a line in the network services database.

The `setservent()` function opens and rewinds the file. If you pass a nonzero `stayopen` argument to `setservent()`, the services database isn’t closed after each call to `getservbyname()` or `getservbyport()`.

The `getservbyname()` and `getservbyport()` functions sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

Returns:

A valid pointer to a `servent` structure, or NULL if an error occurs.
Files:

/etc/services
   Network services database file.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

See also:

endservent(), getprotoent(), getservbyport(), getservent(), servent, setservent()

/etc/services in the Utilities Reference
getservbyport()  
Get a service entry, given a port

Synopsis:

```c
#include <netdb.h>

struct servent * getservbyport( int port,
                                const char * proto );
```

Arguments:

- `port`  The port number for the service.
- `proto` NULL, or the protocol for the service.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getservbyport()` function gets the entry for the given port from the services database, `/etc/services`. This function returns a pointer to a structure of type `servent`, which contains the broken-out fields of a line in the network services database.

The `setservent()` function opens and rewinds the file. If you pass a nonzero `stayopen` argument to `setservent()`, the services database isn’t closed after each call to `getservbyname()` or `getservbyport()`.

The `getservbyport()` function sequentially searches from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

Returns:

A valid pointer to a `servent` structure, or NULL if an error occurs.
**Files:**

`/etc/services`

Network services database file.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**Caveats:**

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

**See also:**

`endservent()`, `getservbyname()`, `getservent()`, `servent`, `setservent()`

`/etc/services` in the *Utilities Reference*
**Synopsis:**

```c
#include <netdb.h>

struct servent * getservent( void );
```

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `getservent()` function reads the next line of network services database file, opening the file if necessary. It returns a pointer to a structure of type `servent`, which contains the broken-out fields of a line in the network services database, `/etc/services`.

**Returns:**

A valid pointer to a `servent` structure, or NULL if an error occurs.

**Files:**

`/etc/services`

Network services database file.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
Caveats:

This function uses static data; if you need the data for future use, copy it before any subsequent calls overwrite it.

See also:

endservent(), getservbyname(), getservbyport(), servent, setservent()

/etc/services in the Utilities Reference
**Synopsis:**

```c
#include <unistd.h>

pid_t getsid( pid_t pid );
```

**Arguments:**

- `pid` The process ID for the process whose session ID you want to get.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `getsid()` function determines the session ID for the given process ID, `pid`.

**Returns:**

The session ID, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EPERM** The process specified by `pid` is not in the same session as the calling process. The implementation doesn’t allow access to the process group ID of the session leader from the calling process.
- **EINVAL** There isn’t a process with the given ID.

**Classification:**

- POSIX 1003.1 XSI
getsid()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt</td>
<td>No</td>
</tr>
<tr>
<td>Signal</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, setsid()
**getsockname()**

Get the name of a socket

**Synopsis:**
```
#include <sys/socket.h>

int getsockname( int s,  
                 struct sockaddr * name,  
                 socklen_t * namelen );
```

**Arguments:**
- `s` The file descriptor of the socket whose name you want to get.
- `name` A pointer to a `sockaddr` object where the function can store the socket’s name.
- `namelen` A pointer to a `socklen_t` object that initially indicates the amount of space pointed to by `name`. The function updates `namelen` to contain the actual size of the name (in bytes).

**Library:**
```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `getsockname()` function returns the current name for the specified socket.

**Returns:**

- 0 Success.
- -1 An error occurred (`errno` is set).
getsockname()

Errors:

EBADF    Invalid descriptor s.
EFAULT   The name parameter points to memory that isn’t in a valid part of the process address space.
ENOBUFFS Insufficient resources were available in the system to perform the operation.

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getpeername()
getsockopt()  
Get options associated with a socket

Synopsis:
```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt( int s,
                int level,
                int optname,
                void * optval,
                socklen_t * optlen);
```

Arguments:

  s  
The file descriptor of the socket that the option is to be applied to, as returned by `socket()`.

  level  
The protocol layer that the option is to be applied to. In most cases, it’s a socket-level option and is indicated by SOL_SOCKET.

  optname  
The option for the socket file descriptor. For a list of options, see “Options,” below.

  optval  
A pointer to the value of the option (in most cases, whether the option is to be turned on or off). If no option value is to be returned, `optval` may be NULL.

Most socket-level options use an `int` parameter for `optval`. Others, such as the SO_LINGER, SO_SNDTIMEO, and SO_RCVTIMEO options, use structures that also let you get data associated with the option.

  optlen  
A pointer to the length of the value of the option. This argument is a value-result parameter; initialize it to indicate the size of the buffer pointed to by `optval`. 
Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `getsockopt()` function gets options associated with a socket.

Manipulating socket options

When manipulating a socket option, you must specify the option’s name (`optname`) and the level (`level`) at which the option resides.

To manipulate options at the socket-level, specify `level` as `SOL_SOCKET`. When manipulating options any other level, the value that you specify for `level` is represented by the protocol number of the appropriate protocol controlling the option. You can obtain the value in a variety of ways:

- from the “Options” section below, use the symbolic constant (e.g. `IPPROTO_IP`, `IPPROTO_TCP`) that corresponds to the option
- from `/etc/protocols`, specify the protocol number for the appropriate protocol
- call `getprotobyname()` and pass the appropriate protocol (e.g. `getprotobyname(tcp)`) to retrieve the number of the protocol level.

The latter two ways might not work if you have customized `/etc/protocols`.

The `optname` parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The `<sys/socket.h>` header file contains definitions for the socket-level options. Options at other protocol levels vary in format and name.
Since levels (e.g. SOL_SOCKET, IPPROTO_IP and IPPROTO_TCP) and the options within the levels can vary, you need to ensure the proper headers are included for both. For example, when setting TCP_NODELAY:

```c
int on = 1;
setsockopt(s, IPPROTO_TCP, TCP_NODELAY, &on);
```
the level IPPROTO_TCP is defined in `<netinet/in.h>`, whereas the TCP_NODELAY option is defined in `<netinet/tcp.h>`.

**Options**

This section describes some of the more common options and their corresponding level.

Except where noted, you can examine the state of the option by calling `getsockopt()`, and set the state by calling `setsockopt()`.

For the list of options that the tiny TCP/IP stack supports, see `npm-ttcpip.so` in the Utilities Reference.

**IP_HDRINCL**

`level`: IPPROTO_IP

Get or set the custom IP header that’s included with your data. You can use it only for raw sockets. For example:

```c
(socket(AF_INET, SOCK_RAW, ...
```

**IP_TOS**

`level`: IPPROTO_IP

Get or set the type-of-service field in the IP header for SOCK_STREAM and SOCK_DGRAM (not applicable for `npm-ttcpip.so`) sockets.
getsockopt()

SO_BINDTODEVICE

`level`: SOL_SOCKET

Applies to `setsockopt()` only.

Allow packets to be sent or received on this specified interface only. If the interface specified conflicts with the parameters of `bind()`, or with the routing table, an error or undesired behavior may occur.

This option accepts the `ifreq` structure with the `ifr_name` member set to the interface name (e.g. `en0`). Currently, you can use this option only for UDP sockets.

SO_BROADCAST

`level`: SOL_SOCKET

Enable or disable the permission to transmit broadcast messages. You can use this option only for UDP sockets. For example:

```c
(socket(AF_INET, SOCK_DGRAM, ...))
```

“Broadcast” was a privileged operation in earlier versions of the system.

SO_DEBUG

`level`: SOL_SOCKET

Enable or disable the recording of debug information in the underlying protocol modules.

SO_DONTROUTE

`level`: SOL_SOCKET

Enable or disable the bypassing of routing tables for outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities. The messages are directed to the appropriate network interface according to the network portion of the destination address.
getsockopt()

SO_ERROR

level: SOL_SOCKET

Applies to getsockopt() only.

Get any pending error on the socket and clears the error status. You can use it to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

SO_KEEPALIVE

level: SOL_SOCKET

Enable or disable the periodic (at least every 2 hours) transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken, and processes that are using the socket are notified via a SIGPIPE signal when they attempt to send data.

SO_LINGER

level: SOL_SOCKET

Controls the action that’s taken when unsent messages are queued on socket when a close() is performed.

If it’s enabled and the socket promises reliable delivery of data, the system blocks the process on the close() attempt until it’s able to transmit the data or until it decides it can’t deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt() call when SO_LINGER is requested).

If it’s disabled, the system processes the close() in a way that lets the process continue as quickly as possible.

The struct linger parameter (defined in <sys/socket.h>) specifies the desired state of the option in the l_onoff field and the linger interval in the l_linger field, in seconds. A value of 0 causes a reset on the socket when the application closes the socket.
SO_OOBINLINE

level: SOL_SOCKET

For protocols that support out-of-band data, allows or disallows out-of-band data to be placed in the normal data input queue as received. The data is accessible using the recv() or read() calls without the MSG_OOB flag. Some protocols always behave as if this option is set.

SO_RCVBUF and SO_SNDBUF

level: SOL_SOCKET

Gets or sets the normal buffer sizes allocated for output (SO_SNDBUF) and input (SO_RCVBUF) buffers. You can increase the buffer size for high-volume connections, or decrease it to limit the possible backlog of incoming data. The system places an absolute limit on these values and defaults them to at least 16K for TCP sockets.

SO_RCVLOWAT

level: SOL_SOCKET

Gets or sets the minimum count for input operations (default is 1). In general, receive calls block until any (nonzero) amount of data is received, and then return with the amount available or the amount requested, whichever is smaller.

If you set the value to be larger than the default, blocking receive calls will wait until they’ve received the low-water mark value or the requested amount, whichever is smaller. Receive calls may still return less than the low-water mark if: an error occurs, a signal is caught, or if the type of data next in the receive queue differs from that returned.

SO_RCVTIMEO

level: SOL_SOCKET

Gets or sets a timeout value for input operations. It accepts a struct timeval parameter (defined in <sys/time.h>) with the
number of seconds and microseconds used to limit waits for input operations to complete.

In the current implementation, this timer is restarted each time additional data is received by the protocol, so the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or, if no data was received, with the error EWOULDBLOCK.

**SO_REUSEADDR**

*level*: SOL_SOCKET

Enables or disables the reuse of duplicate addresses and port bindings. Indicates that the rules used in validating addresses supplied in a `bind()` call allows/disallows local addresses to be reused.

**SO_REUSEPORT**

*level*: SOL_SOCKET

Enables or disables duplicate address and port bindings. Complete duplicate bindings by multiple processes are allowed when they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. See the `reuseport_unicast` option of the `npm-tcip.so` shared object to see how unicast packets are also received on all sockets bound to the same port.

**SO_SNDBWAT**

*level*: SOL_SOCKET

Gets or sets the minimum count for output operations. In BSD, this count is typically 2048, but it is a calculated value in Neutrino. If you require a specific SO_SNDBWAT, you must specify the count. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted (subject to flow control without
blocking), but will process no data if flow control doesn’t allow the
smaller of the low-water mark value or the entire request to be
processed.

A select() operation that tests the ability to write to a socket returns
true only if the low-water mark amount could be processed.

SO_SNDTIMEO

level: SOL_SOCKET

Gets or sets a timeout value for output operations. It accepts a
struct timeval parameter (defined in <sys/time.h>) that
includes the number of seconds and microseconds that are used to
limit waits for output operations to complete. If a send operation has
blocked for this much time, it returns with a partial count or with the
error EWOULDBLOCK if data weren’t sent.

This timer is restarted each time additional data is delivered to the
protocol, implying that the limit applies to output portions ranging in
size from the low-water mark to the high-water mark for output.
Timeouts are restricted to 32 seconds or under.

SO_TIMESTAMP

level: SOL_SOCKET

Enables or disables the reception of a timestamp with datagrams. If
enabled on a SOCK_DGRAM socket, the recvmsg() call returns a
timestamp corresponding to when the datagram was received. The
msg_control field in the msghdr structure points to a buffer that
contains a cmsghdr structure followed by a struct timeval. The
cmsghdr fields have the following values:

cmsg_len = sizeof(struct cmsghdr) + sizeof(struct timeval)
cmsg_level = SOL_SOCKET
cmsg_type = SCM_TIMESTAMP
**getsockopt()**

© 2007, QNX Software Systems GmbH & Co. KG.

**SO_TYPE**

*level*: SOL_SOCKET

Applies to `getsockopt()` only.

Gets the type of the socket (e.g. SOCK_STREAM). This information is useful for servers that inherit sockets on startup.

**SO_USELOOPBACK**

*level*: SOL_SOCKET

Enables or disables the sending process to receive its own routing messages.

**TCP_KEEPALIVE**

*level*: IPPROTO_TCP

Gets or sets the amount of time in seconds between keepalive probes (the default value is 2 hours). It accepts a `struct timeval` parameter with the number of seconds to wait between the keepalive probes.

**TCP_NODELAY**

*level*: IPPROTO_TCP

Don’t delay sending in order to coalesce packets. Under most circumstances, TCP sends data when it’s presented. When outstanding data hasn’t yet been acknowledged, TCP gathers small amounts of output to be sent in a single packet once an acknowledgment is received.

For a few clients (such as windowing systems that send a stream of mouse events that receive no replies), this packetization may cause significant delays. Therefore, TCP provides a boolean option, `TCP_NODELAY`, to defeat this algorithm.
getsockopt()

Returns:

0  Success.
-1  An error occurred (errno is set).

Errors:

EBADF  Invalid file descriptor s.
EDOM   Value was set out of range.
EFAULT The address pointed to by optval isn’t in a valid part of the process address space. For getsockopt(), this error may also be returned if optlen isn’t in a valid part of the process address space.
EINVAL The optval argument can’t be NULL; optlen can’t be 0.
ENOPROTOOPT The option is unknown at the level indicated.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ICMP, IP, TCP, and UDP protocols
getsockopt()

close(), getprotobyname(), ioctl(), read(), select(), setsockopt(),
socket()

/etc/protocols in the Utilities Reference
Synopsis:

```
#include <sys/types.h>
#include <shadow.h>

struct spwd* getspent( void );

struct spwd* getspent_r( struct spwd* result,
                          char* buffer,
                          int buflen );
```

Arguments:

These arguments apply only to `getspent_r()`:

- `result` A pointer to a `spwd` structure where the function can store the entry. For more information about this structure, see `putspent()`.
- `buffer` A block of memory that the function can use to allocate storage referenced by the `spwd` structure. You can determine the maximum size needed for this buffer by calling `sysconf()` with an argument of `_SC_GETPW_R_SIZE_MAX`.
- `bufsize` The size of the block that `buffer` points to, in characters.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getspent()` and `getspent_r()` functions return the next entry from the shadow password database. The `getspent()` function uses a static buffer that’s overwritten by each call.
The `fgetspent()`, `getspent()`, `getspnam()`, and functions share the same static buffer.

Returns:

The `getspent()` function returns a pointer to an object of type `struct spwd` containing the next entry from the shadow password database. When `getspent()` is first called, the database is opened, and remains open until either a NULL is returned to signify end-of-file, or `endspent()` is called.

Errors:

The `getspent()` function uses the following functions, and as a result, `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
- `fopen()`
- `fseek()`
- `rewind()`

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * This program loops, reading a login name from standard
 * input and checking to see if it is a valid name. If it
 * is not valid, the entire contents of the name in the
 * password database are printed.
 */

int main(int argc, char** argv)
{
    struct spwd* sp;
```
char buf[80];

setpwent();
while( gets(buf) != NULL ) {
        if( (sp = getspnam(buf)) != (struct spwd *)0 ) {
                printf("Valid login name is: \%sn", sp->sp_namp);
        } else {
                setspent();
                while( (sp=getspent()) != (struct spwd *)0 )
                        printf("\%sn", sp->sp_namp);
        }
}
endspent();
return( EXIT_SUCCESS );

Classification:
Unix

getspent()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

getspent_r()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
getspent(), getspent_r()

See also:

errno, fgetspent(), getgrent(), getlogin(), getspnam(), getpwuid(),
putspent(), setspent()
Synopsis:

```c
#include <sys/types.h>
#include <shadow.h>

struct spwd* getspnam( char* name );

struct spwd* getspnam_r( const char* name, 
                        struct spwd* result, 
                        char* buffer, 
                        size_t bufsize );
```

Arguments:

- **name**  
  The name of the user.

- **result**  
  (getspnam_r() only) A pointer to a spwd structure where the function can store the entry. For more information about this structure, see putspent().

- **buffer**  
  (getspnam_r() only) A block of memory that the function can use to allocate storage referenced by the spwd structure. You can determine the maximum size needed for this buffer by calling sysconf() with an argument of _SC_GETPW_R_SIZE_MAX.

- **bufsize**  
  (getspnam_r() only) The size of the block that buffer points to, in characters.

Library:

- libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The getspnam() and getspnam_r() functions allow a process to gain more knowledge about a user name. The getspnam() function uses a static buffer that’s overwritten by each call.
The \texttt{fgetspent()}, \texttt{getspent()}, and \texttt{getspnam()} functions share the same static buffer.

Returns:

A pointer to an object of type \texttt{struct spwd} containing an entry from the group database with a matching \textit{name}, or NULL if an error occurred or the function couldn’t find a matching entry.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * Print information from the password entry
 * about the user name given as argv[1].
 */

int main( int argc, char** argv )
{
    struct spwd* sp;
    
    if (argc < 2) {
        printf("\n username \n", argv[0]);
        return(EXIT_FAILURE);
    }

    if( ( sp = getspnam( argv[1] ) ) == (struct spwd*)0) {
        fprintf( stderr, "getspnam: unknown %s
",
            argv[1] );
        return( EXIT_FAILURE );
    }

    printf( "login name %s
", sp->sp_namp );
    printf( "password %s
", sp->sp_pwdp );
    return( EXIT_SUCCESS );
}
```

Classification:

Unix
getsnam()  

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

getsnam_r()  

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:  

fgetspent(), getlogin(), getspent(), getpwuid(), putspent()
getsubopt()  Parse suboptions from a string

Synopsis:

#include <stdlib.h>

int getsubopt( char** optionp,
              char* const* tokens,
              char** valuep );

Arguments:

optionp       The address of a pointer to the string of options that you want to parse. The function updates this pointer as it parses the options; see below.

tokens        A vector of possible tokens.

valuep        The address of a pointer that the function updates to point to the first character of a value that’s associated with an option; see below.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The getsubopt() functions parses suboptions in a flag argument that was initially parsed by getopt(). These suboptions are separated by commas and may consist of either a single token or a token-value pair separated by an equal sign. Since commas delimit suboptions in the option string, they aren’t allowed to be part of the suboption or the value of a suboption. A command that uses this syntax is mount, which allows the user to specify mount parameters with the -o option as follows:

mount -o rw,hard,bg,wsize=1024 speed:/usr /usr
getsubopt()

In this example there are four suboptions: \texttt{rw}, \texttt{hard}, \texttt{bg}, and \texttt{wsize}, the last of which has an associated value of 1024.

The \texttt{getsubopt()} function takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string, or -1 if there was no match. If the option string at \texttt{optionp} contains only one suboption, \texttt{getsubopt()} updates \texttt{optionp} to point to the null character at the end of the string; otherwise, it isolates the suboption by replacing the comma separator with a null character, and updates \texttt{optionp} to point to the start of the next suboption. If the suboption has an associated value, \texttt{getsubopt()} updates \texttt{valuep} to point to the value’s first character. Otherwise, it sets \texttt{valuep} to NULL.

The token vector is organized as a series of pointers to null strings. The end of the token vector is identified by a NULL pointer.

When \texttt{getsubopt()} returns, if \texttt{valuep} isn’t NULL, the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when \texttt{getsubopt()} fails to match the suboption with the tokens in the \texttt{tokens} array, the calling program should decide if this is an error, or if the unrecognized option should be passed to another program.

\textbf{Returns:}

The \texttt{getsubopt()} function returns -1 when the token it’s scanning isn’t in the \texttt{tokens} vector. The variable addressed by \texttt{valuep} contains a pointer to the first character of the token that wasn’t recognized rather than a pointer to a value for that token.

The variable addressed by \texttt{optionp} points to the next option to be parsed, or a null character if there are no more options.
Examples:

The following code fragment shows how to process options to the mount(1M) command using `getsubopt()`:

```c
#include <stdlib.h>

char *myopts[] = {
#define READONLY 0
    "ro",
#define READWRITE 1
    "rw",
#define WRITESIZE 2
    "wsize",
#define READSIZE 3
    "rsize",
    NULL};

main(argc, argv)
    int argc;
    char **argv;
{
    int sc, c, errflag;
    char *options, *value;
    extern char *optarg;
    extern int optind;
    ... ...
    while((c = getopt(argc, argv, "abf:o:")) != -1) {
        switch (c) {
            case 'a': /* process a option */
                break;
            case 'b': /* process b option */
                break;
            case 'f':
                ofile = optarg;
                break;
            case '?':
                errflag++;
                break;
            case 'o':
                options = optarg;
                while (*options != '\0') {
                    switch(getsubopt(&options,myopts,&value)) {
                        case READONLY : /* process ro option */
                            break;
                        case READWRITE : /* process rw option */
                            break;
                        ... ...
                    }
                }
            ... ...
        }
    }
```
case WRITESIZE : /* process wsize option */
    if (value == NULL) {
        error_no_arg();
        errflag++;
    } else
        write_size = atoi(value);
    break;

case READSIZE : /* process rsize option */
    if (value == NULL) {
        error_no_arg();
        errflag++;
    } else
        read_size = atoi(value);
    break;

default :
    /* process unknown token */
    error_bad_token(value);
    errflag++;
    break;
}

break;

if (errflag) {
    /* print usage instructions etc. */
}

for (; optind < argc; optind++) {
    /* process remaining arguments */
    .
    .
    .
}

---

**Classification:**

POSIX 1003.1 XSI

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes

*continued…*
## Caveats:

During parsing, commas in the option input string are changed to null characters. White space in tokens or token-value pairs must be protected from the shell by quotes.

## See also:

`getopt()`
Synopsis:

```c
#include <sys/time.h>

int gettimeofday( struct timeval * when,
                  void * not_used );
```

Arguments:

- `when`: A pointer to a `timeval` structure where the function can store the time. The `struct timeval` contains the following members:
  - `long tv_sec` — the number of seconds since the start of the Unix Epoch.
  - `long tv_usec` — the number of microseconds.
- `not_used`: This pointer must be NULL or the behavior of `gettimeofday()` is unspecified. This argument is provided only for backwards compatibility.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `gettimeofday()` function returns the current time in `when` in seconds and microseconds, since the Unix Epoch, 00:00:00 January 1, 1970 Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

Returns:

0 for success, or -1 if an error occurs (`errno` is set).
gettimeofday() © 2007, QNX Software Systems GmbH & Co. KG.

Errors:

EFAULT  An error occurred while accessing the when buffer.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The gettimeofday() function is provided for porting existing code. You shouldn’t use it in new code; use clock_gettime() instead.

See also:

asctime(), asctime_r(), clock_gettime(), clock_settime(), ctime(), ctime_r(), difftime(), gmtime(), gmtime_r(), localtime(), localtime_r(), settimeofday(), time()
getuid()
Get the user ID

Synopsis:
```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid( void );
```

Library:
```
libc
```
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The getuid() function gets the user ID for the calling process.

Returns:
The user ID for the calling process.

Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    printf( "My userid is %d\n", getuid() );
    return EXIT_SUCCESS;
}
```

Classification:
POSIX 1003.1

Safety
Cancellation point   No
```
continued...
```. 
getuid()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getegid(), geteuid(), getgid(), setuid()
getutent()

Read the next entry from the user-information file

Synopsis:

```c
#include <utmp.h>

struct utmp * getutent( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getutent()` function reads in the next entry from a user-information file. If the file isn’t already open, `getutent()` opens it. If the function reaches the end of the file, it fails.

Returns:

A pointer to a `utmp` structure for the next entry, or NULL if the file couldn’t be read or reached the end of file.

Files:

```c
_PATH_UTMP
```

Specifies the user information file.

Classification:

Unix
getutent()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either `getutid()` or `getutline()`, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it’s searching for, the function looks no further. For this reason, to use `getutline()` to search for multiple occurrences, zero out the static area after each success, or `getutline()` will return the same structure over and over again.

There’s one exception to the rule about emptying the structure before further reads are done: the implicit read done by `pututline()` (if it finds that it isn’t already at the correct place in the file) doesn’t hurt the contents of the static structure returned by the `getutent()`, `getutid()` or `getutline()` routines, if you just modified those contents and passed the pointer back to `pututline()`.

These routines use buffered standard I/O for input, but `pututline()` uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the `utmp` and `wtmp` files.

See also:

`endutent()`, `getutid()`, `getutline()`, `pututline()`, `setutent()`, `utmp`, `utmpname()`

`login` in the Utilities Reference
getutid()
Search for an entry in the user-information file

Synopsis:

```c
#include <utmp.h>

struct utmp * getutid( struct utmp * id );
```

Arguments:

`id`  A pointer to a `utmp` structure that you want to find in the user-information file.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getutid()` function searches forward from the current point in the `utmp` file until it finds a matching entry:

- If `id->ut_type` is one of `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, or `NEW_TIME`, the function looks for an entry with the same `ut_type`.

- If `id->ut_type` is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS`, `getutid()` looks for the first entry with the same `ut_type` and a `ut_id` field that matches `id->ut_id`.

If `getutid()` reaches the end of the file without finding a match, the search fails.
Returns:

A pointer to the `utmp` structure for the matching entry, or NULL if it couldn’t be found.

Files:

`_PATH_UTMP`

Specifies the user information file.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either `getutid()` or `getutline()`, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it’s searching for, the function looks no further. For this reason, to use `getutline()` to search for multiple occurrences, zero out the static area after each success, or `getutline()` will return the same structure over and over again.

There’s one exception to the rule about emptying the structure before further reads are done: the implicit read done by `pututline()` (if it finds that it isn’t already at the correct place in the file) doesn’t hurt the contents of the static structure returned by the `getutent()`, `getutid()` or
getutid() routines, if the user has just modified those contents and passed the pointer back to pututline().

These routines use buffered standard I/O for input, but pututline() uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the utmp and wtmp files.

See also:

endutent(), getutent(), getutline(), pututline(), setutent(), utmp, utmpname()

login in the Utilities Reference
**getutline()**

Get an entry from the user-information file

**Synopsis:**

```c
#include <utmp.h>

struct utmp * getutline( struct utmp * line );
```

**Arguments:**

- `line`  
  A pointer to a `utmp` structure that you want to find in the user-information file.

**Library:**

- `libc`

  Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

  This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `getutline()` function searches forward from the current point in the `utmp` file until it finds an entry of the type LOGIN_PROCESS or a `ut_line` string that matches `line->ut_line`. If the function reaches the end of the file is reached without finding a match, the function fails.

**Returns:**

A pointer to the `utmp` structure for the entry found, or NULL if the search failed.

**Files:**

- `PATH_UTMP`

  Specifies the user information file.
Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either `getutid()` or `getutline()`, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it’s searching for, the function looks no further. For this reason, to use `getutline()` to search for multiple occurrences, zero out the static area after each success, or `getutline()` will return the same structure over and over again.

There’s one exception to the rule about emptying the structure before further reads are done: the implicit read done by `pututline()` (if it finds that it isn’t already at the correct place in the file) doesn’t hurt the contents of the static structure returned by the `getutent()`, `getutid()` or `getutline()` routines, if the user has just modified those contents and passed the pointer back to `pututline()`.

These routines use buffered standard I/O for input, but `pututline()` uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the `utmp` and `wtmp` files.

See also:

`endutent()`, `getutent()`, `getutid()`, `pututline()`, `setutent()`, `utmp`, `utmpname()`
login in the Utilities Reference
getw()

Get a word from a stream

Synopsis:

```c
#include <stdio.h>

int getw( FILE* stream );
```

Arguments:

- `stream` The stream that you want to read a word from.

Library:

- `libc` Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getw()` function returns the next word (i.e. integer) from the named input stream. This function increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer, and varies from machine to machine. The `getw()` function assumes no special alignment in the file.

Returns:

The next word, or the constant EOF at the end-of-file or on an error; it sets the EOF or error indicator of the stream.

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.
Errors:

EOVERFLOW  The file is a regular file, and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

Classification:

Legacy Unix

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Because of possible differences in word length and byte ordering, files written using putw() are implementation-dependent, and might not be read correctly using getw() on a different processor.

See also:

fclose(), feof(), ferror(), fgetc(), flockfile(), fopen(), fread(), getc(), getc_unlocked(), getchar(), getchar_unlocked(), gets(), putc(), putw(), scanf(), ungetc(),
getwc()

Read a wide character from a stream

Synopsis:

```c
#include <wchar.h>

wint_t getwc( FILE * fp );
```

Arguments:

`fp` The stream from which you want to read a wide character.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getwc()` function reads the next wide character from the specified stream.

Returns:

The next character from the stream, cast as `(wint_t)(wchar_t)`, or `WEOF` if end-of-file has been reached or if an error occurs (`errno` is set).

Errors:

- **EAGAIN** The O_NONBLOCK flag is set for `fp` and would have been blocked by this operation.
- **EBADF** The `fp` stream isn’t valid for reading.
- **EINTR** A signal terminated the read operation; no data was transferred.

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.
**getwc()**

Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.

EOVERFLOW Cannot read at or beyond the offset maximum for this stream.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

_errno, feof(), ferror(), putwc(), putwchar()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter
getwchar()

Read a character from a stream

Synopsis:

```c
#include <wchar.h>

wint_t getwchar( void );
```

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `getwchar()` function reads the next wide character from `stdin`.

Returns:

The next character from `stdin`, cast as `(wint_t) (wchar_t)`, or `WEOF` if the end-of-file has been reached or if an error occurs (`errno` is set).

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

Errors:

- **EAGAIN**: The O_NONBLOCK flag is set for `stdin` and would have been blocked by this operation.
- **EINTR**: A signal terminated the read operation; no data was transferred.
- **EIO**: Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.
- **EOVERFLOW**: Cannot read at or beyond the offset maximum for this stream.
getwchar()  © 2007, QNX Software Systems GmbH & Co. KG.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, feof(), ferror(), putwc(), putwchar()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter
getwd()  
Get current working directory pathname

Synopsis:

```c
#include <unistd.h>

char* getwd( char *path_name );
```

Arguments:

- `path_name`: A buffer where the function can store the current working directory.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `getwd()` function determines the absolute pathname of the current working directory of the calling process, and copies that pathname into the array pointed to by the `path_name` argument.

If the length of the pathname of the current working directory is greater than \( \{ \text{PATH_MAX} \} + 1 \) including the null byte, `getwd()` fails and returns a null pointer.

For portability, use `getcwd()` instead of `getwd()`.

Returns:

A pointer to the string containing the absolute pathname of the current working directory. On error, `getwd()` returns a null pointer and the contents of the array pointed to by `path_name` are undefined.
getwd()

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getcwd()
Synopsis:

```
#include <glob.h>

int glob( const char* pattern,
           int flags,
           int (*errfunc)( const char* epath,
                          int error ),
           glob_t* pglob );
```

Arguments:

- **pattern**  The pattern you want to match. This can include these wildcard characters:
  - * matches any string, of any length
  - ? matches any single character
  - [chars] matches any of the characters found in the string *chars*.

- **flags**  Flags that affect the search; see below.

- **errfunc**  A pointer to a function that *glob()* calls when it encounters a directory that it can’t open or read. For more information, see below.

- **pglob**  A pointer to a *glob_t* structure where *glob()* can store the paths found. This structure contains at least the following members:
  - `size_t gl_pathc` — the number of pathnames matched by *pattern*.
  - `char** gl_pathv` — a NULL-terminated array of pointers to the pathnames matched by *pattern*.
  - `size_t gl_offs` — the number of pointers to reserve at the beginning of *gl_pathv*.

You must create the *glob_t* structure before calling *glob()*. The *glob()* function allocates storage as needed for the *gl_pathv* array. Use *globfree()* to free this space.
**Description:**

The `glob()` function finds pathnames matching the given pattern.

In order to have access to a pathname, `glob()` must have search permission on every component of the path except the last, and read permission on each directory of every filename component of `pattern` that contains any of the special characters (`*`, `?`, `[`, and `]`).

The `errfunc` argument is a pointer to an error-handler function with this prototype:

```c
int errfunc( const char* epath, int error );
```

The `errfunc` function is called when `glob()` encounters a directory that it can’t open or read. The arguments are:

- `epath` A pointer to the path that failed.
- `error` The value of `errno` from the failure. The `error` argument can be set to any of the values returned by `opendir()`, `readdir()`, or `stat()`.

The `errfunc` function should return 0 if `glob()` should continue, or a nonzero value if `glob()` should stop searching.

You can set `errfunc` to NULL to ignore these types of errors.

The `flags` argument can be set to any combination of the following bits:
GLOB_APPEND  Append found pathnames to the ones from a previous call from glob().

GLOB_DOOFFS  Use the value in pglob->gl_offs to specify how many NULL pointers to add at the beginning of pglob->pathv. After the call to glob(), pglob->pathv will contain pglob->gl_offs NULL pointers, followed by pglob->gl_pathc pathnames, followed by a NULL pointer. This can be useful if you’re building a command to be applied to the matched files.

GLOB_ERR      Cause glob() to return when it encounters a directory that it can’t open or read. Otherwise, glob() will continue to find matches.

GLOB_MARK     Append a slash to each matching pathname that’s a directory.

GLOB_NOCHECK  If pattern doesn’t match any path names, return only the contents of pattern.

GLOB_NOESCAPE Disable backslash escapes in pattern.

GLOB_NOSORT   Don’t sort the returned pathnames; their order will be unspecified. The default is to sort the pathnames.

Returns:

Zero for success, or an error value.

Errors:

GLOB_ABEND   The scan was stopped because GLOB_ERR was set, or the errmsg function returned nonzero.
GLOB_NOMATCH

The value of pattern doesn’t match any existing pathname, and GLOB_NOCHECK wasn’t set in flags.

GLOB_NOSPACE

Unable to allocate memory to store the matched paths.

Examples:

This simple example attempts to find all of the .c files in the current directory and print them in the order the filesystem found them.

```c
#include <unistd.h>
#include <stdio.h>
#include <glob.h>

int main( void )
{
    glob_t paths;
    int retval;

    paths.gl_pathc = 0;
    paths.gl_pathv = NULL;
    paths.gl_offs = 0;

    retval = glob( "*.c", GLOB_NOCHECK | GLOB_NOSORT, NULL, &paths );
    if( retval == 0 ) {
        int idx;

        for( idx = 0; idx < paths.gl_pathc; idx++ ) {
            printf( "%[d]: %s\n", idx,
                    paths.gl_pathv[idx] );
        }

        globfree( &paths );
    } else {
        puts( "glob() failed" );
    }

    return 0;
}
```
glob()

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

Don’t change the values in pglob between calling glob() and globfree().

See also:

globfree(), wordexp(), wordfree()
**globfree()**

Free storage allocated by a call to glob()

**Synopsis:**

```c
#include <glob.h>

void globfree( glob_t* pglob );
```

**Arguments:**

`pglob` A pointer to a `glob_t` structure that you passed to `glob()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

---

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `globfree()` function frees the storage allocated by a call to `glob()`.

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
globfree()

Caveats:

Don’t change the values in pglob between calling glob() and globfree().

See also:

glob(), wordexp(), wordfree()
gmtime()  
Convert calendar time to a broken-down time

Synopsis:

```c
#include <time.h>

struct tm* gmtime( const time_t* timer );
```

Arguments:

- `timer` A pointer to a `time_t` structure that contains the time that you want to convert.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `gmtime()` function converts the calendar time pointed to by `timer` into a broken-down time, expressed as Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time or GMT).

The `gmtime()` function places the converted time in a static structure that’s reused each time you call `gmtime()` or `localtime()`. Calling `asctime()` or `ctime()` could also change the data in this static buffer.

In a multithreaded application, use `gmtime_r()`.

You typically use the `date` command to set the computer’s internal clock using Coordinated Universal Time (UTC). Use the `TZ` environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.
Returns:

A pointer to the static tm structure that contains the broken-down time.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), clock(), ctime(), difftime(), gmtime_r(), localtime(), localtime_r(), mktime(), strftime(), time(), tm, tzset()

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
gmtime_r()

Convert calendar time to a broken-down time

Synopsis:

```c
#include <time.h>

struct tm* gmtime_r( const time_t* timer,
                      struct tm* result );
```

Arguments:

- `timer` A pointer to a `time_t` structure that contains the time that you want to convert.
- `result` A pointer to a `tm` structure where the function can store the broken-down time.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `gmtime_r()` function converts the calendar time pointed to by `timer` into a broken-down time, expressed as Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time or GMT) and stores it in the `tm` structure pointed to by `result`.

You typically use the `date` command to set the computer’s internal clock using Coordinated Universal Time (UTC). Use the `TZ` environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.

Returns:

A pointer to the `tm` structure containing the broken-down time.
gmtime_r()

Classification:

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), clock(), ctime(), difftime(), localtime(), localtime_r(), mktime(), strftime(), time(), tm, tzset()

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
**Synopsis:**
```
#include <netdb.h>

extern int h_errno;
```

**Library:**

```
libsocket
```
Use the `-l socket` option to `qcc` to link against this library.

**Description:**
The `h_errno` variable can be set by any one of the following functions:

- `gethostbyaddr()`
- `gethostbyaddr_r()`
- `gethostbyname()`
- `gethostbyname2()`
- `gethostbyname_r()`
- `res_query()`
- `res_search()`

It can be set to any one of the following:

- **HOST_NOT_FOUND**
  Authoritative answer: Unknown host.

- **NETDB_INTERNAL**
  You specified an invalid address family when calling `gethostbyname2()`.

- **NO_DATA**
  Valid name, no data record of the requested type.
  The name is known to the name server, but has no
IP address associated with it—this isn’t a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

**NO_RECOVERY**
Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

**TRY_AGAIN**
Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn’t receive a response from an authoritative server. A retry at some later time may succeed.

**Classification:**

POSIX 1003.1 OBS

**Caveats:**

Unlike *errno*, *h_errno* isn’t thread-safe.

**See also:**

*errno, gethostbyaddr(), gethostbyaddr_r(), gethostbyname(), gethostbyname2(), gethostbyname_r(), res_query(), res_search()*
hcreate()  
Create a hash search table

Synopsis:

```
#include <search.h>

int hcreate( size_t nel );
```

Arguments:

`nel`  
An estimate of the maximum number of entries that the table will contain. The algorithm might adjust this number upward in order to obtain certain mathematically favorable circumstances.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `hcreate()` function allocates space for the hash search table. You must call this function before using `hsearch()`.

The `hsearch()` and `hcreate()` functions use `malloc()` to allocate space.

Only one hash search table may be active at any given time. You can destroy the table by calling `hdestroy()`.

Returns:

0 if there isn’t enough space available to allocate the table.

Examples:

See `hsearch()`.
Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

bsearch(), hdestroy(), hsearch(), malloc()

The Art of Computer Programming, Volume 3, Sorting and Searching
Destroy the hash search table

**Synopsis:**

```c
#include <search.h>

void hdestroy( void );
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `hdestroy()` function destroys the hash search table that was created by `hcreate()` and used by `hsearch()`. Only one hash search table may be active at any given time.

**Examples:**

See `hsearch()`.

**Classification:**

POSIX 1003.1 XSI

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

\textit{hcreate()}, \textit{hsearch()}
**Synopsis:**

```c
#include <netdb.h>

void herror( const char* prefix );
```

**Arguments:**

- `prefix`: NULL, or a string that you want to print before the error message.

**Library:**

- **libsocket**
  
  Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `herror()` function prints the message corresponding to the error number contained in `h_errno` to `stderr`. The following functions can set `h_errno`:

- `gethostbyaddr()`
- `gethostbyaddr_r()`
- `gethostbyname()`
- `gethostbyname2()`
- `gethostbyname_r()`
- `res_query()`
- `res_search()`

If the `prefix` string is non-NULL, it’s printed, followed by a colon and a space. The error message is printed with a trailing newline. One of the following messages could be printed:

```
HOST_NOT_FOUND
```

Authoritative answer: Unknown host.
null

You specified an invalid address family when calling `gethostbyname2()`.

Valid name, no data record of the requested type. The name is known to the name server, but has no IP address associated with it—this isn’t a temporary error. Another type of request to the name server using this domain name will result in an answer (e.g. a mail-forwarder may be registered for this domain).

Unknown server error. An unexpected server failure was encountered. This is a nonrecoverable network error.

Nonauthoritative answer: Host name lookup failure. This is usually a temporary error and means that the local server didn’t receive a response from an authoritative server. A retry at some later time may succeed.

**Classification:**

Unix

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

gethostbyaddr(), gethostbyaddr_r(), gethostbyname(),
gethostbyname_r(), h_errno, res_query(), res_search(), stderr
Synopsis:

```
#include <netdb.h>

struct hostent {
    char * h_name;
    char ** h_aliases;
    int h_addrtype;
    int h_length;
    char ** h_addr_list;
};

#define h_addr h_addr_list[0]
```

Description:

This structure describes an Internet host. It contains either the information obtained from a name server, or broken-out fields from a line in /etc/hosts.

The members of this structure are:

- **h_name**: The official name of the host.
- **h_aliases**: A zero-terminated array of alternate names for the host.
- **h_addrtype**: The type of address being returned; currently always AF_INET.
- **h_length**: The length of the address, in bytes.
- **h_addr_list**: A zero-terminated array of network addresses for the host. Host addresses are returned in network byte order.

A **#define** statement is used to define the following:

- **h_addr**: The first address in h_addr_list. This is for backward compatibility.
Classification:
POSIX 1003.1

See also:
endhostent(), gethostbyaddr(), gethostbyname(), gethostent(),
sethostent()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
Synopsis:
#include <search.h>

ENTRY* hsearch ( ENTRY item,
               ACTION action );

Arguments:
item A structure of type ENTRY, defined in <search.h>, that contains:
   - char *key — a pointer to the comparison key.
   - void *data — a pointer to any other data to be associated with the key.

action A member of an enumeration type ACTION, also defined in <search.h>, indicating what to do with the entry if it isn’t in the table:
   - ENTER — insert the entry in the table at the appropriate point. If the item is a duplicate of an existing item, the new item isn’t added, and hsearch() returns a pointer to the existing one.
   - FIND — don’t add the entry. If the item can’t be found, hsearch() returns NULL.

Library:
libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The hsearch() function is a hash-table search routine generalized from Knuth (6.4) Algorithm D. Before using this function, you must call hcreate() to create the hash table.
The `hsearch()` function returns a pointer into a hash table indicating the location at which an entry can be found. This function uses `strcmp()` as the comparison function.

The `hsearch()` and `hcreate()` functions use `malloc()` to allocate space.

Only one hash search table may be active at any given time. You can destroy the table by calling `hdestroy()`.

**Returns:**

A pointer to the item found, or NULL if either the action is FIND and the item wasn’t found, or the action is ENTER and the table is full.

**Examples:**

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings, finds the matching entry in the hash table and prints it.

```c
#include <stdio.h>
#include <search.h>
#include <string.h>
#include <stdlib.h>

struct info { /* this is the info stored in table */
    int age, room; /* other than the key */
};
#define NUM_EMPL 5000 /* # of elements in search table */

main( )
{
    /* space to store strings */
    char string_space[NUM_EMPL*20];
    /* space to store employee info */
    struct info info_space[NUM_EMPL];
    /* next avail space in string_space */
    char *str_ptr = string_space;
    /* next avail space in info_space */
    struct info *info_ptr = info_space;

    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d\d", str_ptr, &info_ptr->age,
                   &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* name to look for in table */
        char name_to_find[30];
        int i = 0;

        /* enter item */
        item = hinsert(str_ptr, info_ptr, FIND);
        if (item != NULL)
            strcpy(name_to_find, str_ptr);
        else
            name_to_find[0] = 0;

        /* find name */
        item = hsearch(str_ptr, FIND);
        if (item != NULL)
            printf("%s\n", info_ptr->name);
        else
            printf("%s not found\n", name_to_find);
    }
}
```
hsearch()

/* put info in structure, and structure in item */
item.key = str_ptr;
item.data = (void *)info_ptr;
str_ptr += strlen(str_ptr) + 1;
info_ptr++;
/* put item into table */
(void) hsearch(item, ENTER);
}

/* access table */
item.key = name_to_find;
while (scanf("%s", item.key) != EOF) {
    if ((found_item = hsearch(item, FIND)) != NULL) {
        /* if item is in the table */
        (void)printf("found %s, age = %d, room = %d\n",
            (struct info *)found_item->data)->age,
            (struct info *)found_item->data)->room);
    } else {
        (void)printf("no such employee %s\n",
            name_to_find);
    }
}

hdestroy();
return 0;

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`bsearch()`, `hcreate()`, `hdestroy()`, `malloc()`, `strcmp()`

*The Art of Computer Programming, Volume 3, Sorting and Searching*
Synopsis:

```
#include <netdb.h>

const char* hstrerror( int err );
```

Arguments:

- **err**  
The error code that you want to get the message for. For more information, see `h_errno`.

Library:

```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `hstrerror()` function gets an error message string associated with the error return status from network host-related functions.

Network host-related functions such as the following can return the error status:

- `gethostbyaddr()`, `gethostbyaddr_r()`  
- `gethostbyname()`, `gethostbyname_r()`  
- `res_query()`  
- `res_search()`

You can check the external integer `h_errno` to see whether this is a temporary failure or an invalid or unknown host.

Returns:

A pointer to the message string affiliated with an error number.
Don’t modify the message string.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`h_errno`, `herror`
htonl()

Convert a 32-bit value from host-byte order to network-byte order

Synopsis:

```c
#include <arpa/inet.h>

uint32_t htonl( uint32_t hostlong);
```

Arguments:

- `hostlong` The value that you want to convert.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `htonl()` function converts a 32-bit value from host-byte order to network-byte order.

You typically use this routine in conjunction with the internet addresses and ports that `gethostbyname()` and `getservent()` return.

Returns:

The value in network-byte order.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`gethostbyname()`, `getservent()`, `htons()`, `ntohl()`, ` ntohs()`
Synopsis:

```c
#include <arpa/inet.h>

uint16_t htons( uint16_t hostshort );
```

Arguments:

`hostshort`  The value that you want to convert.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `htons()` function converts a 16-bit value from host-byte order to network-byte order.

You typically use this routine in conjunction with the internet addresses and ports that `gethostbyname()` and `getservent()` return.

Returns:

The value in network-byte order.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

gethostbyname(), getservent(), htonl(), ntohl(), ntohs()
Synopsis:

```c
#include <hw/sysinfo.h>

unsigned hwi_find_item( unsigned start,
                        ... );
```

Arguments:

- **start**
  Where to start the search for the given item.
  For the initial call, set this argument to HWI_NULL_OFF.
  If the item found isn’t the one that you want, pass the
  return value from the first call to `hwi_find_item()` as the
  `start` parameter of the next call. This makes the search pick
  up where it left off. You can repeat this process as many
  times as required (the return value from the second call
  going into the `start` parameter of the third, etc).

- **char ***
  A sequence of names for identifying the item being
  searched.
  Terminate the sequence with a NULL pointer. The last
  string before the NULL is the bottom-level item name that
  you’re looking for, the string in front of that is the name of
  the item that owns the bottom-level item, etc.

Library:

- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is
  usually included automatically.

---

This function is in `libc.a`, but not in `libc.so` (in order to save space).
**hwi_find_item()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The `hwi_find_item()` function finds an item in the `hwinfo` structure of the system page.

**Returns:**

The offset of the item requested, or `HWI_NULL_OFF` if the item wasn’t found.

**Examples:**

Find the first occurrence of an item called “foobar”:

```c
item_off = hwi_find_item(HWI_NULL_OFF, "foobar", NULL);
```

Find the first occurrence of an item called “foobar” that’s owned by “sam”:

```c
item_off = hwi_find_item(HWI_NULL_OFF, "sam", "foobar", NULL);
```

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `hwi_find_tag()`, `hwi_off2tag()`, `hwi_tag2off()`
- “Structure of the system page” in the Customizing Image Startup Programs chapter of *Building Embedded Systems*
**Synopsis:**

```c
#include <hw/sysinfo.h>

unsigned hwi_find_tag( unsigned start, 
                       int curr_item, 
                       const char * tagname );
```

**Arguments:**

- `start`: Where to start to search for the given item.
  For the initial call, set this argument to HWI_NULL_OFF. If the item found isn’t the one that you want, pass the return value from the first call to `hwi_find_tag()` as the `start` parameter of the next call. This makes the search pick up where it left off. You can repeat this process as many times as required (the return value from the second call going into the `start` parameter of the third, etc).

- `curr_item`: If this argument is nonzero, the search stops at the end of the current item (i.e. the one that `start` points to). If `curr_item` is zero, the search continues until the end of the section.

- `tagname`: The name of tag to search for.

**Library:**

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).
Description:

The `hwi_find_tag()` function finds the tag named `tagname`.

Returns:

The offset of the tag, or `HWI_NULL_OFF` if the tag wasn’t found.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `hwi_find_item()`, `hwi_off2tag()`, `hwi_tag2off()`
- “Structure of the system page” in the Customizing Image Startup Programs chapter of *Building Embedded Systems*
hwi_off2tag()

Return a pointer to the start of a tag in the hwinfo area of the system page

Synopsis:

```c
#include <hw/sysinfo.h>

void * hwi_off2tag( unsigned offsect );
```

Arguments:

- `offsect`: The offset, in bytes from the start of the `hwinfo` section, of a tag.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `hwi_off2tag()` function returns a pointer to the start of the tag, given an offset.

Returns:

A pointer to the start of the tag.

Classification:

- QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: Yes

continued...
**hwi_off2tag()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`hwi_find_item(), hwi_find_tag(), hwi_tag2off()`

“Structure of the system page” in the Customizing Image Startup Programs chapter of *Building Embedded Systems*
Synopsis:

#include <hw/sysinfo.h>

unsigned hwi_tag2off( void *tag );

Arguments:

tag  A pointer to a tag in the hwinfo area of the system page.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).

Description:

Given a pointer to the start of a tag, the hwi_tag2off() function returns the offset, in bytes, from the beginning of the start of the hwinfo section.

Returns:

The offset of the tag, in bytes.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
hwi_tag2off()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

hwi_find_item(), hwi_find_tag(), hwi_off2tag()

“Structure of the system page” in the Customizing Image Startup Programs chapter of Building Embedded Systems
**Synopsis:**

```c
#include <math.h>

double hypot( double x, double y );

float hypotf( float x, float y );
```

**Arguments:**

- `x`, `y`  The lengths of the sides that are adjacent to the right angle.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

These functions compute the length of the hypotenuse for a right triangle whose sides are `x` and `y` adjacent to the right angle. The calculation is equivalent to:

```c
length = sqrt( x*x + y*y );
```

**Returns:**

The length of the hypotenuse.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", hypot( 3.0, 4.0 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
5.000000
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`sqrt`
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td><code>abort()</code> to <code>expm1f()</code></td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td><code>fabs()</code> to <code>hypotf()</code></td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to <code>lround()</code></td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td><code>main()</code> to <code>outle32()</code></td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td><code>pathconf()</code> to <code>ruserok()</code></td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td><code>sbrk()</code> to <code>system()</code></td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td><code>tan()</code> to <code>ynf()</code></td>
</tr>
</tbody>
</table>
Internet Control Message Protocol

Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_RAW,
            proto );
```

Description:

ICMP is the error- and control-message protocol used by IP and the Internet protocol family. The protocol may be accessed through a “raw socket” for network monitoring and diagnostic functions.

To get the `proto` parameter to `socket()` that’s used to create an ICMP socket, call `getprotobyname()`. You normally use ICMP sockets, which are connectionless, with `sendto()` and `recvfrom()`, although you can also use `connect()` to fix the destination for future packets (in which case you can use the `read()` or `recv()`, and `write()` or `send()` system calls).

Outgoing packets automatically have an IP header prepended to them that’s based on the destination address. Incoming packets are received with the IP header and IP options intact.

Returns:

A descriptor referencing the socket, or -1 if an error occurs (`errno` is set).

Errors:

- **EADDRNOTAVAIL**
  
  Tried to create a socket with a network address for which no network interface exists.

- **EISCONN**
  
  Tried to establish a connection on a socket that already has one, or tried to send a datagram with the destination address specified but the socket is already connected.
ICMP

ENOBUS – The system ran out of memory for an internal data structure.

ENOTCONN – Tried to send a datagram, but no destination address was specified, and the socket hasn’t been connected.

See also:

ICMP6, IP protocols

connect(), getprotobynamel, read(), recv(), recvfrom(), send(), sendto(), socket(), write()

RFC 792
ICMP6
© 2007, QNX Software Systems GmbH & Co. KG.

Internet Control Message Protocol for IP6

Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/icmp6.h>

int socket( AF_INET6,
            SOCK_RAW,
            proto);
```

Description:

ICMP6 is the error and control message protocol that IP6 and the Internet Protocol family use. It may be accessed through a “raw socket” for network monitoring and diagnostic functions. Use the `getprotobyname()` function to obtain the `proto` parameter to the `socket()` function, or simply pass `IPPROTO_ICMPV6`.

ICMPv6 sockets are connectionless, and are normally used with the `sendto()` and `recvfrom()` functions. You may also use the `connect()` function to fix the destination for future packets (in which case, you may also use the `read()` or `recv()` functions and the `write()` or `send()` system calls).

Outgoing packets automatically have an IP6 header prepended to them (based on the destination address). The ICMP6 pseudo header checksum field (`icmp6_cksum`, found in the `icmp6_hdr` structure in `<netinet/icmp6.h>`) is filled automatically by the socket manager. Incoming packets are received without the IP6 header or extension headers.

---

This behavior is opposite from both IPv4 raw sockets and ICMPv4 sockets.

---

ICMP6 type/code filter

Each ICMP6 raw socket has an associated filter whose data type is defined as `struct icmp6_filter`. This structure, along with the macros and constants defined below are defined in the `<netinet/icmp6.h>` header.
You can get and set the current filter by calling `getsockopt()` and `setsockopt()` with a level of IPPROTO_ICMPV6 and an option name of ICMP6_FILTER.

The following macros operate on an `icmp6_filter` structure. If the first argument is an integer, it represents an ICMP6 message type, with a value between 0 and 255. The pointer arguments are pointers to the filters that are either set or examined, depending on the macro:

```
ICMP6_FILTER_SETPASSALL( struct icmp6_filter* )
   Pass all ICMPv6 messages to the application.

ICMP6_FILTER_SETBLOCKALL( struct icmp6_filter* )
   Block all ICMPv6 messages from the application.

ICMP6_FILTER_SETPASS( int, struct icmp6_filter* )
   Pass messages of a certain ICMPv6 type to the application.

ICMP6_FILTER_SETBLOCK( int, struct icmp6_filter* )
   Block messages of a certain ICMPv6 type from the application.

ICMP6_FILTER_WILLPASS( int, const struct icmp6_filter* )
   Return true or false, depending on whether or not the specified message type is passed to the application.

ICMP6_FILTER_WILLBLOCK( int, const struct icmp6_filter* )
   Return true or false, depending on whether or not the specified message type is blocked from the application.
```

When you create an ICMP6 raw socket, it passes all ICMPv6 message types to the application by default.

For more information, see RFC 2292.
See also:

INET6, IP6 protocols

`connect()`, `getprotobynname()`, `getsockopt()`, `read()`, `recv()`, `recvfrom()`, `send()`, `sendto()`, `setsockopt()`, `socket()`, `write()`

`RFC 2292`
Synopsis:

```c
#include <net/if.h>

void if_freenameindex( struct if_nameindex * ptr );
```

Arguments:

`ptr` A pointer to the `if_nameindex` structure to be freed.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `if_freenameindex()` function frees the dynamic memory that you allocated by calling `if_nameindex()`.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`getifaddrs()`, `if_indextoname()`, `if_nameindex()`, `if_nametoindex()`
if_indextoname()
Map an interface index to its name

Synopsis:

```c
#include <net/if.h>

char * if_indextoname( unsigned int ifindex,
            char * ifname );
```

Arguments:

- `ifindex`  The interface index.
- `ifname`   A pointer to a buffer in which `if_indextoname()` copies the interface name. The buffer must be a minimum of IFNAMSIZ bytes long.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `if_indextoname()` function maps the interface index specified by `ifindex` to its corresponding name. The name is copied into the buffer pointed to by `ifname`.

Returns:

A pointer to the name, or NULL if there isn’t an interface corresponding to the specified index.

Classification:

- POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

continued...
### if_indextoname()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

getifaddrs(), if_frenameindex(), if_nameindex(), if_nametoindex()
if_nameindex()

Return a list of interfaces

Synopsis:

```c
#include <net/if.h>

struct if_nameindex * if_nameindex( void );
```

Library:

```c
libsocket
```

Use the -l socket option to qcc to link against this library.

Description:

The `if_nameindex()` function returns an array of `if_nameindex` structures, with one structure per interface, as defined in the include file `<net/if.h>`. The `if_nameindex` structure contains at least the following members:

- `unsigned int if_index`
  - The index of the interface (1, 2, ...).
- `char *if_name`
  - A null-terminated name (e.g. `le0`).

The end of the array of structures is indicated by an entry with an `if_index` of 0 and an `if_name` of NULL.

Returns:

A valid array of `if_nameindex` structures, or NULL if and error occurred while using `getifaddrs()` to retrieve the list, or there wasn’t enough memory available.

Classification:

POSIX 1003.1
### if_nameindex()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*getifaddrs(), if_freenameindex(), if_indextoname(), if_nametoindex()*
**if nametoindex()**

Map an interface name to its index

**Synopsis:**

```c
#include <net/if.h>

unsigned int if_nametoindex( const char * ifname );
```

**Arguments:**

- `ifname` The interface name that you want to map.

**Library:**

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `if_nametoindex()` function maps the interface name specified by `ifname` to its corresponding index.

**Returns:**

The index number of the interface, or 0 if the specified interface couldn’t be found or an error occurred while using `getifaddrs()` to retrieve the list of interfaces.

**Classification:**

POSIX 1003.1

**Safety**

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
See also:

getifaddrs(), if_freenameindex(), if_indextoname(), if_nameindex()
ifaddrs
Structure that describes an Internet host

Synopsis:

```c
#include <ifaddrs.h>

struct ifaddrs {
    struct ifaddrs * ifa_next;
    char * ifa_name;
    u_int ifa_flags;
    struct sockaddr * ifa_addr;
    struct sockaddr * ifa_netmask;
    struct sockaddr * ifa_dstaddr;
    void * ifa_data;
};
```

Description:

The `ifaddrs` structure contains the following entries:

- `ifa_next`  
  A pointer to the next structure in the list. This field is NULL in the last structure in the list.

- `ifa_name`  
  The interface name.

- `ifa_flags`  
  The interface flags, as set by the `ifconfig` utility.

- `ifa_addr`  
  Either the address of the interface or the link-level address of the interface, if one exists; otherwise it’s NULL. See the `sa_family` member of the `sockaddr` structure pointed to by `ifa_addr` to determine the format of the address.

- `ifa_netmask`  
  The netmask associated with `ifa_addr`, if one is set; otherwise it’s NULL.

- `ifa_dstaddr`  
  The destination address on a P2P interface, if one exists; otherwise it’s NULL. If the interface isn’t a P2P interface, `ifa_dstaddr` contains the broadcast address associated with `ifa_addr`, if one exists; otherwise it’s NULL (see `<ifaddrs.h>`).

- `ifa_data`  
  Currently, this is set to NULL.
Classification:

Unix

See also:

freeifaddrs(), getifaddrs()
Synopsis:

```c
#include <math.h>

int ilogb ( double x );
int ilogbf ( float x );
```

Arguments:

- `x` The number you want to compute the integral part of the logarithm.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `ilogb()` and `ilogbf()` functions compute the integral part of:

\[ \log_r |x| \]

as a signed integral value, for nonzero finite `x`, where `r` is the radix of the machine’s floating point arithmetic.

Returns:

The exponent part of `x`, in integer format:

<table>
<thead>
<tr>
<th>If <code>x</code> is:</th>
<th><code>ilogb()</code> returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-INT_MAX</td>
</tr>
<tr>
<td>NAN</td>
<td>INT_MAX</td>
</tr>
<tr>
<td>negative infinity</td>
<td>INT_MAX</td>
</tr>
<tr>
<td>positive infinity</td>
<td>INT_MAX</td>
</tr>
</tbody>
</table>
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", ilogb(.5) );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`log(), logb(), log10(), log1p()`
Synopsis:

```c
#include <hw/inout.h>

uint8_t in8( uintptr_t port );
```

Arguments:

`port`  The port you want to read the value from.

Library:

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `in8()` function reads an 8-bit value from the specified `port`.

Returns:

An 8-bit value.

Classification:

- QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The calling thread must have I/O privileges; see ThreadCtl()’s
_NTO_TCTL_IO command for details.

The calling process must also use mmap_device_io() to access the
device’s I/O registers.

See also:

in8s(), in16(), in16s(), in32(), in32s(), mmap_device_io(), out8(),
out8s(), out16(), out16s(), out32(), out32s()
Synopsis:

```c
#include <hw/inout.h>

void * in8s( void * buff,
             unsigned len,
             uintptr_t port );
```

Arguments:

- **buff**: A pointer to a buffer where the function can store the values read.
- **len**: The number of values that you want to read.
- **port**: The port you want to read the values from.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `in8s()` function reads `len` 8-bit values from the specified `port` and stores them in the buffer pointed to by `buff`.

Returns:

A pointer to the end of the read data.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>continued...</td>
<td></td>
</tr>
</tbody>
</table>
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The calling thread must have I/O privileges; see `ThreadCtl()`’s `_NTO_TCTL_IO` command for details.

The calling process must also use `mmap_device_ios()` to access the device’s I/O registers.

**See also:**

`in8()`, `in16()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_ios()`, `out8()`, `out8s()`, `out16()`, `out16s()`, `out32()`, `out32s()`
Synopsis:

```c
#include <hw/inout.h>

uint16_t in16( uintptr_t port );
#define inbe16 ( port ) ...
#define inle16 ( port ) ...
```

Arguments:

- `port` The port you want to read the value from.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `in16()` function reads a 16-bit value from the specified `port` in native-endian format (there’s no conversion required).

The `inbe16()` and `inle16()` macros read a 16-bit value that’s in big-endian or little-endian format, respectively, from the specified `port`, and returns the value as native-endian.

Returns:

A 16-bit value in native-endian.

Classification:

QNX Neutrino
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The calling thread must have I/O privileges; see `ThreadCtl()`'s `_NTO_TCTL_IO` command for details.

The calling process must also use `mmap_device_io()` to access the device’s I/O registers.

Both `inbe16()` and `inle16()` are implemented as macros.

See also:

`in8()`, `in8s()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_io()`, `out8()`, `out8s()`, `out16()`, `out16s()`, `out32()`, `out32s()`
Synopsis:

```c
#include <hw/inout.h>

void * in16s( void * buff,
              unsigned len,
              uintptr_t port );
```

Arguments:
- `buff`: A pointer to a buffer where the function can store the values read.
- `len`: The number of values that you want to read.
- `port`: The port you want to read the values from.

Library:
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `in16s()` function reads `len` 16-bit values from the specified `port` and stores them in the buffer pointed to by `buff`.

Returns:

A pointer to the end of the read data.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

continued...
**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The calling thread must have I/O privileges; see `ThreadCtl()`’s 
`_NTO_TCTL_IO` command for details.

The calling process must also use `mmap_device_io()` to access the device’s I/O registers.

**See also:**

`in8()`, `in8s()`, `in16()`, `in32()`, `in32s()`, `mmap_device_io()`, `out8()`, `out8s()`, `out16()`, `out16s()`, `out32()`, `out32s()`
**Synopsis:**

```c
#include <hw/inout.h>

uint32_t in32( uintptr_t port );

#define inbe32 ( port ) ...

#define inle32 ( port ) ...
```

**Arguments:**

- `port` The port you want to read the value from.

**Library:**

- `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `in32()` function reads a 32-bit value from the specified `port`.

The `inbe32()` and `inle32()` macros read a 32-bit value that’s in big-endian or little-endian format, respectively, from the specified `port`, and returns the value as native-endian.

**Returns:**

A 32-bit value in native-endian.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

*continued...*
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The calling thread must have I/O privileges; see `ThreadCtl()`’s \_NTO\_TCTL\_IO command for details.

The calling process must also use `mmap_device_io()` to access the device’s I/O registers.

Both `inbe32()` and `inle32()` are implemented as macros.

**See also:**

`in8()`, `in8s()`, `in16()`, `in16s()`, `in32s()`, `mmap_device_io()`, `out8()`, `out8s()`, `out16()`, `out16s()`, `out32()`, `out32s()`
### Synopsis:

```c
#include <hw/inout.h>

void * in32s( void * buff,
              unsigned len,
              uintptr_t port);
```

### Arguments:

- **buff**: A pointer to a buffer where the function can store the values read.
- **len**: The number of values that you want to read.
- **port**: The port you want to read the values from.

### Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `in32s()` function reads `len` 32-bit values from the specified `port` and stores them in the buffer pointed to by `buff`.

### Returns:

A pointer to the end of the read data.

### Classification:

QNX Neutrino

---

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
</table>

*continued...*
in32s()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The calling thread must have I/O privileges; see ThreadCtl()’s _NTO_TCTL_IOC command for details.

The calling process must also use mmap_device_io() to access the device’s I/O registers.

See also:

in8(), in8s(), in16(), in16s(), in32(), mmap_device_io(), out8(), out8s(), out16(), out16s(), out32(), out32s()
Synopsis:

```c
#include <strings.h>

char* index( const char* s, int c );
```

Arguments:

- `s`: The string you want to search. This string must end with a null (\0) character. The null character is considered to be part of the string.
- `c`: The character you’re looking for.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `index()` function returns a pointer to the first occurrence of the character `c` in the string `s`.

Returns:

A pointer to the character, or NULL if the character doesn’t occur in the string.

Classification:

POSIX 1003.1 XSI

Safety

- Cancellation point: No

continued…
index()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

rindex(), strchr(), strrchr()
**inet_addr()**

Convert a string into a numeric Internet address

**Synopsis:**

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
```

**Arguments:**

- `cp` A pointer to a string that represents an Internet address.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `inet_addr()` routine converts a string representing an IPv4 Internet address (for example, "127.0.0.1") into a numeric Internet address. To convert a hostname such as `ftp.qnx.com`, call `gethostbyname()`.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see `inet_net_ntop()`.

**Returns:**

An Internet address, or `INADDR_NONE` if an error occurs.

**Classification:**

POSIX 1003.1

**Safety**

Cancellation point: No

continued...
Inet_addr()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Although the value INADDR_NONE (0xFFFFFFFF) is a valid broadcast address, inet_addr() always indicates failure when returning that value. The inet_aton() function doesn’t share this problem.

See also:

inet_aton(), inet_network()
**inet_aton()**

*Convert a string into an Internet address stored in a structure*

**Synopsis:**

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton( const char * cp,
               struct in_addr * addr );
```

**Arguments:**

- `cp` A pointer to the character string.
- `addr` A pointer to a `in_addr` structure where the function can store the converted address.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `inet_aton()` routine interprets the specified character string as an IPv4 Internet address, placing the address into the structure provided.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values.

For more information on Internet addresses, see `inet_net_ntop()`.

**Returns:**

- 1 Success; the string was successfully interpreted.
- 0 Failure; the string is invalid.
inet_aton() © 2007, QNX Software Systems GmbH & Co. KG.

Classification:
Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

gethostbyname(), getnetent(), inet_addr(), inet_lnaof(),
inet_makeaddr(), inet_netoif(), inet_network(), inet_ntoa()

/etc/hosts, /etc/networks in the Utilities Reference
inet_lnaof()

Extract the local network address from an Internet address

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_lnaof( struct in_addr in );
```

Arguments:

in 
An Internet address.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The `inet_lnaof()` routine returns the local network address for an IPv4 Internet address.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see `inet_net_ntop()`.

Returns:

A local network address.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>continued...</td>
<td></td>
</tr>
</tbody>
</table>
**inet_Lnaof()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`inet_aton()`, `inet_netof()`
Convert a network number and a local network address into an Internet address

Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr( unsigned long net,
                               unsigned long lna );
```

Arguments:

- `net` An Internet network number.
- `lna` The local network address.

Library:

- `libsocket`
  
  Use the `-l socket` option to `qcc` to link against this library.

Description:

The `inet_makeaddr()` routine takes an Internet network number and a local network address and constructs an IPv4 Internet address.

All Internet addresses are returned in network byte order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see `inet_net_ntop()`.

Returns:

An Internet address.

Classification:

QNX Neutrino
**inet_makeaddr()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*inet_aton()*
inet_net_ntop()

Convert an Internet network number to CIDR format

Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_net_ntop( int af,
    const void * src,
    int bits,
    char * dst,
    size_t size );
```

Arguments:

- `af` The address family. Currently, only AF_INET is supported.
- `src` A pointer to the Internet network number that you want to convert. The format of the address is interpreted according to `af`.
- `bits` The number of bits that specify the network number (`src`).
- `dst` A pointer to the buffer where the function can store the converted address.
- `size` The size of the buffer that `dst` points to, in bytes.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `inet_net_ntop()` function converts an Internet network number from network format (usually a `struct in_addr` or some other binary form, in network byte order) to CIDR (Classless Internet Domain Routing) presentation format that’s suitable for external display purposes.

With CIDR, a single IP address can be used to designate many unique IP addresses. A CIDR IP address looks like a normal IP address,
except that it ends with a slash (/) followed by a number, called the IP prefix. For example:

172.200.0.0/16

The IP prefix specifies how many addresses are covered by the CIDR address, with lower numbers covering more addresses.

Network Numbers (IPv4 Internet addresses)

You can specify Internet addresses in the “dotted quad” notation, or Internet network numbers, using one of the following forms:

\( a.b.c.d\) or \( a.b.c.d\)\( /\)\( bits\)

When you specify a four-part address, each part is interpreted as a byte of data and is assigned, from left to right, to the four bytes of an Internet network number (or Internet address). When an Internet network number is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (i.e. right to left), such as the Intel 386, 486 and Pentium processors, the bytes referred to above appear as “d.c.b.a”.

\( a.b.c\)

When you specify a three-part address, the last part is interpreted as a 16-bit quantity and is placed in the rightmost two bytes of the Internet network number (or network address). This makes the three-part address format convenient for specifying Class B network addresses as \( net.net.host\).

\( a.b\)

When you specify a two-part address, the last part is interpreted as a 24-bit quantity and is placed in the rightmost three bytes of the Internet network number (or network address). This makes the two-part number format convenient for specifying Class A network numbers as \( net.host\).

\( a\)

When you specify a one-part address, the value is stored directly in the Internet network number (network address) without any byte rearrangement.
All numbers supplied as “parts” in a dot notation may be decimal, octal, or hexadecimal, as specified in the C language. That is, a number is interpreted as decimal unless it has a leading 0 (octal), or a leading 0x or 0X (hex).

**Returns:**

A pointer to the destination string (`dst`), or NULL if a system error occurs (`errno` is set).

**Errors:**

ENOENT Invalid argument `af`.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`inet_aton()`, `inet_net_ntop()`
**inet_netof()**

Extract the network number from an Internet address

**Synopsis:**

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_netof( struct in_addr in );
```

**Arguments:**

- `in` An Internet address.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `inet_netof()` routine returns the network number of the specified IPv4 Internet address.

All Internet addresses are returned in network order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see `inet_net_ntop()`.

**Returns:**

An Internet network number.

**Classification:**

QNX Neutrino

**Safety**

Cancellation point  No

continued…
See also:

inet_aton(), inet_lnaofo()
**inet_net_pton()**

Convert an Internet network number from CIDR format to network format

**Synopsis:**
```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_net_pton( int af,
                    const char * src,
                    void * dst,
                    size_t size );
```

**Arguments:**
- `af` The address family. Currently, only AF_INET is supported.
- `src` A pointer to the presentation-format (CIDR) address. The format of the address is interpreted according to `af`.
- `dst` A pointer to the buffer where the function can store the converted address.
- `size` The size of the buffer pointed to by `dst`, in bytes.

**Library:**
```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `inet_net_pton()` function converts an Internet network number from presentation format — a printable form as held in a character string, such as, Internet standard dot notation, or Classless Internet Domain Routing (CIDR) — to network format (usually a struct `in_addr` or some other internal binary representation, in network byte order).

For more information on Internet addresses, see `inet_net_ntop()`.
Returns:

The number of bits that specify the network number (computed based on the class, or specified with /CIDR), or -1 if an error occurred (errno is set).

Errors:

ENOENT Invalid argument af.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

inet_aton(), inet_net_ntop()
inet network() — Convert a string into an Internet network number

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network( const char * cp );
```

Arguments:

*cp* A pointer to a string representing an Internet address.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The *inet_network()* routine converts a string representing an IPv4 Internet address (for example, "127.0.0.1") into a numeric Internet network number.

All Internet addresses are returned in network order (bytes are ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. For more information on Internet addresses, see *inet_net_ntop()*.

Returns:

An Internet network number, or INADDR_NONE if an error occurs.

Classification:

QNX Neutrino

Safety

Cancellation point No

continued…
**inet_network()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`inet_addr()`, `inet_aton()`
Convert an Internet address into a string

**Synopsis:**
```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char * inet_ntoa( struct in_addr in );
```

**Arguments:**
- `in` The Internet address that you want to convert.

**Library:**
- libsocket

Use the `-l socket` option to `qcc` to link against this library.

**Description:**
The `inet_ntoa()` routine converts an IPv4 Internet address into an ASCII string representing the address in dot notation (for example, “127.0.0.1”).

For more information on Internet addresses, see `inet_net_ntop()`.

**Returns:**
A string representing an Internet address.

**Classification:**
POSIX 1003.1

**Safety**
- Cancellation point: No
- Interrupt handler: No
- Signal handler: No

*continued...*
## Caveats:

The string returned by this function is stored in a static buffer that’s reused for every call to `inet_ntoa()`. For a thread-safe version, see `inet_ntoa_r()`.

### See also:

`inet_aton()`, `inet_ntoa_r()`
**inet_ntoa_r()**

Convert an Internet address into a string

**Synopsis:**

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char * inet_ntoa_r( struct in_addr in, char * buffer, int bufflen );
```

**Arguments:**

- `in` The Internet address that you want to convert.
- `buffer` A buffer where the function can store the result.
- `bufflen` The size of the buffer, in bytes.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `inet_ntoa_r()` function is a thread-safe version of `inet_ntoa()`. It converts an IPv4 Internet address into a string (for example, “127.0.0.1”). For more information on this routine, see `inet_aton()`.

**Returns:**

A string representing an Internet address, or NULL if an error occurs (`errno` is set).

**Errors:**

- `ERANGE` The supplied `buffer` isn’t large enough to store the result.
**inet_ntoa_r()**

**Classification:**

Unix

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*inet_aton(), inet_ntoa()*
Synopsis:
```
#include <sys/socket.h>
#include <arpa/inet.h>

const char * inet_ntop( int af,
                       const void * src,
                       char * dst,
                       socklen_t size );
```

Arguments:
- `af` The `src` address’s network family; one of:
  - AF_INET IPv4 addresses
  - AF_INET6 IPv6 addresses
- `src` The numeric network address that you want to convert to a string.
- `dst` The text string that represents the translated network address. You can use the following constants to allocate buffers of the correct size (they’re defined in `<netinet/in.h>`):
  - INET_ADDRSTRLEN — storage for an IPv4 address
  - INET6_ADDRSTRLEN — storage for an IPv6 address
- `size` The size of the buffer pointed to by `dst`.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `inet_ntop()` function converts a numeric network address pointed to by `src` into a text string in the buffer pointed to by `dst`.

Convert a numeric network address to a string

**inet_ntop()**

© 2007, QNX Software Systems GmbH & Co. KG.
inet_ntop()

Returns:
A pointer to the buffer containing the text version of the address, or NULL if an error occurs (errno is set).

Errors:
EAFNOSUPPORT
The value of the af argument isn’t a supported network family.
ENOSPC The dst buffer isn’t large enough (according to size) to store the translated address.

Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>

#define INADDR "10.1.0.29"
#define IN6ADDR "DEAD:BEEF:7654:3210:FE8C:3210:7654:BA98"

int main()
{
    struct in_addr inaddr;
    struct in6_addr in6addr;
    char buf[INET_ADDRSTRLEN], buf6[INET6_ADDRSTRLEN];
    int rval;

    if ( (rval = inet_pton(AF_INET, INADDR, &inaddr)) == 0) {
        printf("Invalid address: %s\n", INADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(AF_INET, &inaddr, buf, sizeof(buf)) != NULL)
        printf("inet addr: %s\n", buf);
    else {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }
}
```
if ( (rval = inet_pton(AF_INET6, IN6ADDR, &in6addr)) == 0) {
    printf("Invalid address: %s\n", IN6ADDR);
    exit(EXIT_FAILURE);
} else if (rval == -1) {
    perror("inet_pton");
    exit(EXIT_FAILURE);
}

if (inet_ntop(AF_INET6, &in6addr, buf6, sizeof(buf6)) != NULL)
    printf("inet6 addr: %s\n", buf6);
else {
    perror("inet_ntop");
    exit(EXIT_FAILURE);
}

return(EXIT_SUCCESS);

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

inet_ntop()
inet_pton()

Convert a text host address to a numeric address

Synopsis:
```c
#include <sys/socket.h>
#include <arpa/inet.h>

int inet_pton( int af, 
               const char * src, 
               void * dst );
```

Arguments:
- `af` The `src` address’s network family; one of:
  - AF_INET  IPv4 addresses
  - AF_INET6 IPv6 addresses
- `src` A pointer to the text host address that you want to convert. The format of the address is interpreted according to `af`
- `dst` A pointer to a buffer where the function can store the converted address.

Library:
- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `inet_pton()` function converts the standard text representation of the numeric network address (`src`) into its numeric network byte-order binary form (`dst`).

The converted address is stored in network byte order in `dst`. The buffer pointed to by `dst` must be large enough to hold the numeric address:
### inet_pton()

<table>
<thead>
<tr>
<th>Family</th>
<th>Numeric address size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_INET</td>
<td>4 bytes</td>
</tr>
<tr>
<td>AF_INET6</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

#### AF_INET addresses

IPv4 addresses must be specified in the standard dotted-decimal form:

```
ddd.ddd.ddd.ddd
```

where `ddd` is a one- to three-digit decimal number between 0 and 255.

Many existing implementations of `inet_addr()` and `inet_aton()` accept nonstandard input: octal numbers, hexadecimal numbers, and fewer than four numbers. The `inet_pton()` function doesn’t accept these formats.

#### AF_INET6 addresses

IPv6 addresses must be specified in one of the following standard formats:

- The preferred form is:
  
  `x:x:x:x:x:x:x:x`
  
  where `x` is a hexadecimal value for one of the eight 16-bit pieces of the address. For example:

  ```
  417A:200C:800:8:0:0:0:1080
  ```

- `A::` can be used once per address to represent multiple groups of 16 zero-bits. For example, the following addresses:

  ```
  1080:0:0:0:8:800:200C:417A
  FE01:0:0:0:0:0:0:43
  0:0:0:0:0:0:0:1
  0:0:0:0:0:0:0:0
  ```

  can be represented as:
A convenient format when dealing with mixed IPv4 and IPv6 environments is:
\[ x:x:x:x:x:d.d.d \]
where \( x \) is a hexadecimal value for one of the six high-order 16-bit pieces of the address and \( d \) is a decimal value for one of the four low-order 8-bit pieces of the address (standard AF_INET representation). For example:

- 0:0:0:0:0:13.1.68.3
- 0:0:0:0:FFFF:129.144.52.38

Or, in their compressed forms:

- ::13.1.68.3
- ::FFFF:129.144.52.38

**Returns:**

- 1 Success.
- 0 The input isn’t a valid address.
- -1 An error occurred (\texttt{errno} is set).

**Errors:**

- EAFNOSUPPORT
  
  The \texttt{af} argument isn’t one of the supported networking families.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>

#define INADDR "10.1.0.29"
```

int main()
{
    struct in_addr inaddr;
    struct in6_addr in6addr;
    char buf[INET_ADDRSTRLEN], buf6[INET6_ADDRSTRLEN];
    int rval;

    if ( (rval = inet_pton(AF_INET, INADDR, &inaddr)) == 0) {
        printf("Invalid address: %s\n", INADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(AF_INET, &inaddr, buf, sizeof(buf)) != NULL)
        printf("inet addr: %s\n", buf);
    else {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }

    if ( (rval = inet_pton(AF_INET6, IN6ADDR, &in6addr)) == 0) {
        printf("Invalid address: %s\n", IN6ADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_pton");
        exit(EXIT_FAILURE);
    }

    if (inet_ntop(AF_INET6, &in6addr, buf6, sizeof(buf6)) != NULL)
        printf("inet6 addr: %s\n", buf6);
    else {
        perror("inet_ntop");
        exit(EXIT_FAILURE);
    }

    return(EXIT_SUCCESS);
}
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*inet_ntop()*

*RFC 2373*
Synopsis:

```c
#include <netinet/in.h>

struct sockaddr_in6 {
    uint8_t sin6_len;
    sa_family_t sin6_family;
    in_port_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
};
```

Description:

Protocols

The INET6 family consists of the:

- IPv6 network protocol
- Internet Control Message Protocol version 6 (ICMP)
- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP).

TCP supports the SOCK_STREAM abstraction, while UDP supports the SOCK_DGRAM abstraction. Note that TCP and UDP are common to INET and INET6. A raw interface to IPv6 is available by creating an Internet SOCK_RAW socket. The ICMPv6 message protocol may be accessed from a raw socket.

The INET6 protocol family is an updated version of the INET family. While INET implements Internet Protocol version 4, INET6 implements Internet Protocol version 6.

Addressing

IPv6 addresses are 16-byte quantities, stored in network standard (big-endian) byte order. The header file `<netinet/in.h>` defines this address as a discriminated union.

Sockets bound to the INET6 family use the structure shown above.
You can create sockets with the local address :: (which is equal to IPv6 address 0:0:0:0:0:0:0:0) to cause “wildcard” matching on incoming messages. You can specify the address in a call to `connect()` or `sendto()` as :: to mean the local host. You can get the :: value by setting the `sin6_addr` field to 0, or by using the address contained in the `in6addr_any` global variable, which is declared in `<netinet/in6.h>`.

The IPv6 specification defines scoped addresses, such as link-local or site-local addresses. A scoped address is ambiguous to the kernel if it’s specified without a scope identifier. To manipulate scoped addresses properly in your application, use the advanced API defined in RFC 2292. A compact description on the advanced API is available in IP6. If you specify scoped addresses without an explicit scope, the socket manager may return an error.

Scoped addresses are currently experimental, from both a specification and an implementation point of view.

The KAME implementation supports extended numeric IPv6 address notation for link-local addresses. For example, you can use `fe80::1%de0` to specify “fe80::1 on the de0 interface.” The `getaddrinfo()` and `getnameinfo()` functions support this notation. Some utilities, such as `telnet` and `ftp`, can use the notation. With special programs like `ping6`, you can disambiguate scoped addresses by specifying the outgoing interface with extra command-line options.

The socket manager handles scoped addresses in a special manner. In the socket manager’s routing tables or interface structures, a scoped address’s interface index is embedded in the address. Therefore, the address contained in some of the socket manager structures isn’t the same as on the wire. The embedded index becomes visible when using the `PF_ROUTE` socket or the `sysctl()` function. You shouldn’t use the embedded form.
Interaction between IPv4/v6 sockets

The behavior of the AF_INET6 TCP/UDP socket is documented in the RFC 2553 specification, which states:

- A specific bind on an AF_INET6 socket (bind() with an address specified) should accept IPv6 traffic to that address only.

- If you perform a wildcard bind on an AF_INET6 socket (bind() to the IPv6 address ::), and there isn’t a wildcard-bound AF_INET socket on that TCP/UDP port, then the IPv6 traffic as well as the IPv4 traffic should be routed to that AF_INET6 socket. IPv4 traffic should be seen by the application as if it came from an IPv6 address such as ::ffff:10.1.1.1. This is called an IPv4 mapped address.

- If there are both wildcard-bound AF_INET sockets and wildcard-bound AF_INET6 sockets on one TCP/UDP port, they should operate independently: IPv4 traffic should be routed to the AF_INET socket, and IPv6 should be routed to the AF_INET6 socket.

However, the RFC 2553 specification doesn’t define the constraint between the binding order, nor how the IPv4 TCP/UDP port numbers and the IPv6 TCP/UDP port numbers relate each other (whether they must be integrated or separated). The behavior is very different from implementation to implementation. It is unwise to rely too much on the behavior of the AF_INET6 wildcard-bound socket. Instead, connect to two sockets, one for AF_INET and another for AF_INET6, when you want to accept both IPv4 and IPv6 traffic.

---

**CAUTION:** Use caution when handling connections from IPv4 mapped addresses with AF_INET6 sockets—if the target node routes IPv4 traffic to AF_INET6 sockets, malicious parties can bypass security.

Because of the security hole, by default, NetBSD doesn’t route IPv4 traffic to AF_INET6 sockets. If you want to accept both IPv4 and IPv6 traffic, use two sockets. IPv4 traffic may be routed with multiple
per-socket/per-node configurations, but it isn’t recommended. See IP6 for details.

The IPv6 support is subject to change as the Internet protocols develop. Don’t depend on details of the current implementation, but rather the services exported. Try to implement version-independent code as much as possible, because you’ll need to support both INET and INET6.

See also:

ICMP, ICMP6, IP6, IP, TCP, UDP protocols
bind(), connect(), getaddrinfo(), ioctl(), sendto(), socket(), sysctl()

ftp, ping6, telnet in the Utilities Reference

RFC 2553, RFC 2292
inet6_option_alloc() © 2007, QNX Software Systems GmbH & Co. KG.
Append IPv6 hop-by-hop or destination options into ancillary data object

Synopsis:
#include <netinet/in.h>

u_int8_t * inet6_option_alloc(struct cmsghdr *cmsg,
    int datalen,
    int multx,
    int plusy);

Arguments:

  cmsg    A pointer to the cmsghdr structure that must have been initialized by inet6_option_init().
  datalen The length of the option, in bytes. This value is required as an argument to allow the function to determine if padding should be appended at the end of the option, argument since the option data length must already be stored by the caller (the inet6_option_append() function doesn’t need a data length).
  multx   The value x in the alignment term xn + y. It must have a value of 1, 2, 4, or 8.
  plusy   Value y in the alignment term xn + y. It must have a value between 0 and 7, inclusive.

Library:

libsocket
Use the -l socket option to qcc to link against this library.

Description:

This inet6_option_alloc() function appends a hop-by-hop option or a destination option into an ancillary data object that has been initialized by inet6_option_init().
The difference between this function and `inet6_option_append()` is that the latter copies the contents of the previously built option into the ancillary data object. This function returns a pointer to the space in the data object where the option’s type-length-value or TLV must then be built by the caller.

**Returns:**

A pointer to the 8-bit option type field that starts the option, or NULL if an error has occurred.

**Classification:**

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`inet6_option_append()`, `inet6_option_find()`, `inet6_option_init()`, `inet6_option_next()`, `inet6_option_space()`


inet6_option_append()© 2007, QNX Software Systems GmbH & Co. KG.
Append an IPv6 hop-by-hop or destination option to an ancillary data object.

Synopsis:
```
#include <netinet/in.h>

int inet6_option_append(struct cmsghdr *cmsg,
                        const u_int8_t *typep,
                        int multx,
                        int plusy);
```

Arguments:
- **cmsg**: A pointer to the `cmsghdr` structure that must have been initialized by `inet6_option_init()`.
- **typep**: A pointer to the 8-bit option type. It’s assumed that this field is immediately followed by the 8-bit option data length field, which is then followed by the option data. You must initialize these three fields (the type-length-value, or TLV) before calling this function.
  - The option type must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the Pad1 and PadN options, respectively.)
  - The option data length must be between 0 and 255, inclusive, and is the length of the option data that follows.
- **multx**: The value `x` in the alignment term `xn + y`. It must have a value of 1, 2, 4, or 8.
- **plusy**: The value `y` in the alignment term `xn + y`. It must have a value between 0 and 7, inclusive.

Library:
```
libsocket
```
Use the `-l socket` option to `qcc` to link against this library.
Description:

This `inet6_option_append()` function appends a hop-by-hop option or a destination option to an ancillary data object that has been initialized by `inet6_option_init()`.

Returns:

0    Success.

-1   An error has occurred.

Classification:

RFC 2292

See also:

`inet6_option_alloc()`, `inet6_option_find()`, `inet6_option_init()`, `inet6_option_next()`, `inet6_option_space()`


inet6_option_find()

Search for IPv6 hop-by-hop and destination options

Synopsis:

```c
#include <netinet/in.h>

int inet6_option_find(const struct cmsghdr *cmsg,
                      u_int8_t **tptrp,
                      int type );
```

Arguments:

- `cmsg` A pointer to the `cmsghdr` structure that must have been initialized by `inet6_option_init()`.
- `type` The type of option to search for. Either IPV6_HOPOPTS or IPV6_DSTOPTS. This type is stored in the `cmsg_type` member of the `cmsghdr` structure pointed to by `*cmsgp`.
- `tptrp` A pointer to a pointer to an 8-bit byte.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

This `inet6_option_find()` function is similar to `inet6_option_next()`. It however, lets the caller specify the option type to be searched for, instead of always returning the next option in the ancillary data object. The `cmsg` is a pointer to the `cmsghdr` structure of which `cmsg_level` equals IPPROTO_IPV6 and `cmsg_type` equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

The `tptrp` is a pointer to a pointer to an 8-bit byte that the function uses to remember its place in the ancillary data object each time the function is called.

The first time you call this function for a given ancillary data object, you must set `*tptrp` must be set to NULL. This function starts...
searching for an option of the specified type beginning after the value of *tptrp pointer.

**Returns:**

0 with *tptrp pointing to the 8-bit option
   The option was found.
-1 with *tptrp pointing to NULL
   The option wasn’t found.
-1 with *tptrp pointing to non-NULL
   An error has occurred.

**Classification:**

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

*inet6_option_alloc(), inet6_option_append(), inet6_option_init(),
inet6_option_next(), inet6_option_space()*


**inet6_option_init()**

Initialize an ancillary data object that contains IPv6 hop-by-hop and destination options

**Synopsis:**

```
#include <netinet/in.h>

int inet6_option_init(void *bp,
                      struct cmsghdr **cmsgp,
                      int type);
```

**Arguments:**

- **bp**
  A pointer to previously allocated space that contains the ancillary data object. It must be large enough to contain all the individual options to be added by later calls to `inet6_option_append()` and `inet6_option_alloc()`.

- **cmsgp**
  A pointer to a `cmsghdr` structure. The *cmsgp* variable is initialized by this function to point to the `cmsghdr` structure that this function constructs in the buffer pointed to by *bp*.

- **type**
  The type of option which must be either IPV6_HOPOPTS or IPV6_DSTOPTS. This type is stored in the `cmsg_type` member of the `cmsghdr` structure pointed to by *cmsgp*.

**Library:**

```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

Call `inet6_option_init()` function once per ancillary data object that contains either hop-by-hop or destination options.
Returns:

0       Success.
-1      An error has occurred.

Classification:

RFC 2292

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

*inet6_option_alloc(), inet6_option_append(), inet6_option_find(),
inet6_option_next(), inet6_option_space(*


**inet6_option_next()**

Find the next IPv6 hop-by-hop or destination option

**Synopsis:**

```c
#include <netinet/in.h>

int inet6_option_next(const struct cmsghdr *cmsg, u_int8_t **tptrp);
```

**Arguments:**

- `cmsg` A pointer to the `cmsghdr` structure that must have been initialized by `inet6_option_init()`.
- `tptrp` A pointer to a pointer to an 8-bit byte.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

This `inet6_option_next()` function finds the next hop-by-hop option or destination option in an ancillary data object. If another option remains to be processed, the return value of the function is 0 and `tptrp` points to the 8-bit option type field the option data.

The `cmsg` variable is a pointer to `cmsghdr` structure for which `cmsg_level` equals IPPROTO_IPV6 and `cmsg_type` equals either IPV6_HOPOPTS or IPV6_DSTOPTS.

The `tptrp` is a pointer to a pointer to an 8-bit byte and `tptrp` is used by the function to remember its place in the ancillary data object each time the function is called. The first time you call this function for a given ancillary data object, you must set `tptrp` to NULL.

Each time this function returns success, `tptrp` points to the 8-bit option type field for the next option to be processed.
Returns:

0    The option is located and the *tptrp points to the 8-bit option type field.
-1 with *tptrp pointing to NULL
       No more options to process.
-1 with *tptrp pointing to non-NULL
       An error has occurred.

Classification:

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

inet6_option_alloc(), inet6_option_append(), inet6_option_find(),
inet6_option_init(), inet6_option_space()


**inet6_option_space()**

Determine how much space an IPv6 hop-by-hop or destination option requires

**Synopsis:**
```c
#include <netinet/in.h>

int inet6_option_space(int nbytes);
```

**Arguments:**
- `nbytes` The size of the structure that defines the option. It includes any padding bytes at the beginning (the value y in the alignment term \( xn + y \), the type byte), the length byte, and the option data.

**Library:**
- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**
This `inet6_option_space()` function returns the number of bytes required to hold an option when it’s stored as ancillary data, including the `cmsghdr` structure at the beginning, and any padding at the end (to make its size a multiple of 8 bytes).

---

When multiple options are stored in a single ancillary data object, this function overestimates the amount of space required by the size of \( N-1 \) `cmsghdr` structures, where \( N \) is the number of options to be stored in the object. This is of little consequence, since it’s assumed that most hop-by-hop option and destination option headers carry only one option (see Appendix B of RFC 2460).
**Classification:**

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*inet6_option_alloc(), inet6_option_append() inet6_option_find(), inet6_option_init(), inet6_option_next()*


inet6_rthdr_add()

Add an address to an IPv6 routing header

Synopsis:

```c
#include <netinet/in.h>

int inet6_rthdr_add(struct cmsghdr *cmsg,
                       const struct in6_addr *addr,
                       unsigned int flags);
```

Arguments:

- **addr**: A pointer to the IPv6 address structure to add to the routing header.
- **flags**: Routing header flags. For an IPv6 Type 0 routing header, it’s either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT.
- **cmsg**: A pointer to Ancillary data containing the routing header.

Library:

libsocket

Use the `-l socket` option to `qcc` to link against this library.

Description:

This function adds the address pointed to by `addr` to the end of the Routing header being constructed and sets the type of this hop to the value of `flags`.

If successful, the `cmsg_len` member of the `cmsghdr` structure is updated to account for the new address in the routing header.

Returns:

- **0**: Success.
- **-1**: An error has occurred.
**inet6_rthdr_add()**

**Classification:**

RFC 2292

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`inet6_rthdr_getaddr()`, `inet6_rthdr_getflags()`, `inet6_rthdr_init()`, `inet6_rthdr_lasthop()`, `inet6_rthdr_reverse()`, `inet6_rthdr_segments()`, `inet6_rthdr_space()`


**inet6_rthdr_getaddr()**

*Get pointer to an IPv6 address in the routing header*

**Synopsis:**

```c
#include <netinet/in.h>

struct in6_addr * inet6_rthdr_getaddr(
    struct cmsghdr *cmsg,
    int index);
```

**Arguments:**

- `cmsg`: A pointer to the Ancillary data containing the routing header.
- `index`: A value between 0 and the number returned by `inet6_rthdr_segments()`.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

This function returns a pointer to the IPv6 address specified by `index` in the routing header described by `cmsg`. The `index` must have a value between 1 and the number returned by `inet6_rthdr_segments()`. You should first call `inet6_rthdr_segments()` to obtain the number of segments in the Routing header.

**Returns:**

A pointer to the IPv6 address, or NULL if an error occurred.

**Classification:**

RFC 2292
**inet6_rthdr_getaddr()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

- `inet6_rthdr_add()`, `inet6_rthdr_getflags()`, `inet6_rthdr_init()`, `inet6_rthdr_lasthop()`, `inet6_rthdr_reverse()`, `inet6_rthdr_segments()`, `inet6_rthdr_space()`


**inet6_rthdr_getflags()**  
Get the flags for a segment in an IPv6 routing header

**Synopsis:**

```
#include <netinet/in.h>

int inet6_rthdr_getflags(const struct cmsghdr *cmsg,
                         int index);
```

**Arguments:**

- `cmsg` A pointer to the Ancillary data containing the routing header.
- `index` A value between 0 and the number returned by `inet6_rthdr_segments()`.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

This function returns the flags for the segment specified by `index` in the routing header described by `cmsg`. The `index` must have a value between 0 and the number returned by `inet6_rthdr_segments()`.

Addresses are indexed starting at 1, and flags starting at 0. They’re consistent with the terminology and figures in RFC2460.

**Returns:**

- `IPV6_RTHDR_LOOSE` or `IPV6_RTHDR_STRICT` for an IPv6 Type 0 routing header
- `-1` on error.
**Classification:**

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`inet6_rthdr_add()`, `inet6_rthdr_getaddr()`, `inet6_rthdr_init()`, `inet6_rthdr_lasthop()`, `inet6_rthdr_reverse()`, `inet6_rthdr_segments()`, `inet6_rthdr_space()`


**Synopsis:**

```c
#include <netinet/in.h>

struct cmsghdr * inet6_rthdr_init(void *bp,
int type);
```

**Arguments:**

- `bp` A pointer to the buffer where the function can build a `cmsghdr` structure followed by a Routing header of the specified type.

- `type` The type of IPv6 Routing header (e.g. Type 0 as defined in `<netinet/in.h>`).

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

This function initializes the buffer pointed to by `bp` to contain a `cmsghdr` structure followed by a Routing header of the specified type. The `cmsg_len` member of the `cmsghdr` structure is initialized to the size of the structure plus the amount of space required by the Routing header.

The `cmsg_level` and `cmsg_type` members are also initialized as required.

You must allocate the buffer before calling this function. To determine the size of the buffer, call `inet6_rthdr_space()`.
Returns:

A pointer to the `cmsghdr` structure, which you’ll pass to other functions (and used as the first argument to list functions) or NULL if an error occurred.

Classification:

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`inet6_rthdr_add()`, `inet6_rthdr_getaddr()`, `inet6_rthdr_getflags()`,
`inet6_rthdr_lasthop()`, `inet6_rthdr_reverse()`,
`inet6_rthdr_segments()`, `inet6_rthdr_space()`


**inet6_rthdr_lasthop()**

Specify the Strict/Loose flag for the final hop of an IPv6 routing header

**Synopsis:**

```
#include <netinet/in.h>

int inet6_rthdr_lasthop(struct cmsghdr *cmsg,
                        unsigned int flags);
```

**Arguments:**

- **cmsg** Ancillary data containing routing header.
- **flags** Routing header flags. It’s either IPV6_RTHDR_LOOSE or IPV6_RTHDR_STRICT for an IPv6 Type 0 routing header.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

This function specifies the Strict/Loose flag for the final hop of a Routing header.

---

A routing header specifying \( N \) intermediate nodes requires \( N+1 \) Strict/Loose flags. This requires \( N \) calls to `inet6_rthdr_add()` followed by one call to `inet6_rthdr_lasthop()`.

**Returns:**

- **0** Success.
- **-1** An error has occurred.
inet6_rthdr_lasthop()  

Classification:

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

inet6_rthdr_add(), inet6_rthdr_getaddr(), inet6_rthdr_getflags(), inet6_rthdr_init(), inet6_rthdr_reverse(), inet6_rthdr_segments(), inet6_rthdr_space()


**inet6_rthdr_reverse()**

Reverse the list of addresses in an IPv6 router header

**Synopsis:**

```c
#include <netinet/in.h>

int inet6_rthdr_reverse(const struct cmsghdr *in,
                         struct cmsghdr *out);
```

**Arguments:**

- `in` Ancillary data containing Routing header.
- `out` Ancillary data containing Routing header.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `inet6_rthdr_reverse()` has not been implemented yet.

This function takes a routing header that has been received as ancillary data (pointed to by the first argument, `in`) and writes a new routing header. The routing header sends datagrams along the reverse of that route. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

**Returns:**

- `0` Success.
- `-1` An error has occurred.
Classification:

RFC 2292

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

*inet6_rthdr_add()*, *inet6_rthdr_getaddr()*, *inet6_rthdr_getflags()*,
*inet6_rthdr_init()*, *inet6_rthdr_lasthop()*, *inet6_rthdr_segments()*,
*inet6_rthdr_space()*


**inet6_rthdr_segments()**

Count the segments in an IPv6 routing header

**Synopsis:**

```c
#include <netinet/in.h>

int inet6_rthdr_segments(const struct cmsghdr *cmsg);
```

**Arguments:**

- `cmsg`       A pointer to Ancillary data containing a routing header.

**Library:**

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

`inet6_rthdr_segments()`

This function returns the number of segments (addresses) contained in the Routing header described by `cmsg`.

**Returns:**

- 1 to 23  Success.
- -1        An error has occurred.

**Classification:**

- RFC 2292

**Safety**

- Cancellation point  No

*continued...*
**inet6_rthdr_segments()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`inet6_rthdr_add()`, `inet6_rthdr_getaddr()`, `inet6_rthdr_getflags()`,
`inet6_rthdr_init()`, `inet6_rthdr_lasthop()`, `inet6_rthdr_reverse()`,
`inet6_rthdr_space()`


Determine the space required by an IPv6 routing header

Synopsis:

```c
#include <netinet/in.h>

size_t inet6_rthdr_space(int type,
                          int segments);
```

Arguments:

- **type**
  - The type of IPv6 Routing header (e.g. Type 0 as defined in `<netinet/in.h>`).

- **segments**
  - The number of segments (addresses) in the Routing header.

Library:

- **libsocket**
  Use the `-l socket` option to `qcc` to link against this library.

Description:

This function returns the number of bytes required to hold a Routing header of the specified type containing a specified number of segments (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 1 and 23, inclusive. The return value includes the size of the `cmsghdr` structure that precedes the Routing header, and any required padding.

This function returns the size but doesn’t allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired. All the ancillary data objects must be specified to `sendmsg()` as a single `msg_control` buffer in the `msghdr` structure `msg_control` member.
Returns:

0, for either of the two situations: the type of the routing header isn’t supported by this implementation or the number of segments is invalid for this type of routing header.

Classification:

RFC 2292

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

inet6_rthdr_add(), inet6_rthdr_getaddr(), inet6_rthdr_getflags(), inet6_rthdr_init(), inet6_rthdr_lasthop(), inet6_rthdr_reverse(), inet6_rthdr_segments(), inet6_rthdr_space()


**Synopsis:**

```c
#include <grp.h>
#include <sys/types.h>

int initgroups( const char * name,  
gid_t basegid );
```

**Arguments:**

- `name` The name of the user whose group membership you want to use as the supplementary group access list.
- `basegid` A group ID that you want to include in the group access list.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `initgroups()` function reads the group membership for the user specified by `name` from the group database, and then initializes the supplementary group access list of the calling process (see `getgrnam()` and `getgroups()`).

If the number of groups in the supplementary access list exceeds `NGROUPS_MAX`, the extra groups are ignored.

**Returns:**

- `0` Success.
- `-1` An error occurred (`errno` is set).
initgroups()

Errors:

EPERM The caller isn’t root.

Files:

/etc/group The group database.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

If initgroups() fails, it doesn’t change the supplementary group access list.

The getgrouplist() function called by initgroups() is based on getgrent(). If the calling process uses getgrent(), the in-memory group structure is overwritten in the call to initgroups().

See also:

groups(), getgrnam()
**initstate()**  
© 2007, QNX Software Systems GmbH & Co. KG.

Initialize a pseudo-random number generator

**Synopsis:**

```c
#include <stdlib.h>

char* initstate(unsigned int seed, char* state, size_t size);
```

**Arguments:**

- **seed**  
  A starting point for the random-number sequence. This lets you restart the sequence at the same point.

- **state**  
  The state array that you want to initialize.

- **size**  
  The size, in bytes, of the state array; see below.

**Library:**

- **libc**  
  Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

---

**Description:**

The `initstate()` initializes the given state array for future use when generating pseudo-random numbers.

This function uses the `size` argument to determine what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. For values smaller than 8, `random()` uses a simple linear congruential random number generator.

Use this function in conjunction with the following:
initstate()

`random()` Generate a pseudo-random number using a default state.

`setstate()` Specify the state of the pseudo-random number generator.

`srandom()` Set the seed used by the pseudo-random number generator.

If you haven’t called `initstate()`, `random()` behaves as though you had called `initstate()` with a seed of 1 and a size of 128.

After initialization, you can restart a state array at a different point in one of these ways:

- Call `initstate()` with the desired seed, state array, and size of the array.

- Call `setstate()` with the desired state, then call `srandom()` with the desired seed. The advantage of using both of these functions is that the size of the state array doesn’t have to be saved once it’s initialized.

Returns:

A pointer to the previous state array, or NULL if an error occurred.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

static char state1[32];

int main() {
    initstate( time(NULL), state1, sizeof(state1));
    setstate(state1);
    printf("%d
", random());
    return EXIT_SUCCESS;
}
```
Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

drand48(), rand(), random(), setstate(), srand(), srandom()
input_line()

Get a string of characters from a file

Synopsis:

```c
#include <stdio.h>

char* input_line( FILE* fp,
                   char* buf ,
                   int bufsize ) ;

extern int _input_line_max ;
```

Arguments:

- **fp**  The file that you want to read from.
- **buf** A pointer to a buffer where the function can store the 
  string that it reads.
- **bufsize** The size of the buffer, in bytes.

Library:

**libc**

Use the -l c option to qcc to link against this library. This library is 
usually included automatically.

This function is in libc.a, but not in libc.so (in order to save 
space).

Description:

The `input_line()` function gets a string of characters from the file 
designated by `fp` and stores them in the array pointed to by `buf`. The 
`input_line()` function stops reading characters when:

- end-of-file is reached
- a newline character is read
- `bufsize` - 1 characters have been read.
In addition, the `input_line()` function buffers the last `_input_line_max` lines internally. The `_input_line_max` variable is defined in `<stdio.h>`. You can set it before calling `input_line()` for the first time; its default value is 20. While the line is being read, the KEY_UP and KEY_DOWN keys can be used to move to the previous and next line respectively in a circular buffer of previously read lines. The newline character (`\n`) is replaced with the null character on input.

**Returns:**

A pointer to the input line. On end-of-file or on encountering an error reading from `fp`, NULL is returned and `errno` is set.

**Examples:**

```c
#include <stdlib.h>
#include <stdio.h>
#define SIZ 256

int _input_line_max;

int main( void )
{
    FILE  *fp;
    char  *p,
          buf[SIZ];

    fp = stdin; /* Or any stream */
    _input_line_max = 25; /* set before 1st call */

    while( ( p = input_line( fp, buf, SIZ ) ) != NULL ) {
        printf( "%s\n", buf );
        fflush( stdout );
    }
    return EXIT_SUCCESS;
}
```

**Classification:**

QNX 4
## Safety

<table>
<thead>
<tr>
<th>Safety Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
InterruptAttach(), InterruptAttach_r()

Synopsis:

```c
#include <sys/neutrino.h>

int InterruptAttach( int intr,
    const struct sigevent * (* handler)(void *, int),
    const void * area,
    int size,
    unsigned flags );

int InterruptAttach_r( int intr,
    const struct sigevent * (* handler)(void *, int),
    const void * area,
    int size,
    unsigned flags );
```

Arguments:

- **intr**: The interrupt that you want to attach a handler to; see “Interrupt vector numbers,” below.
- **handler**: A pointer to the handler function; see “Interrupt handler function,” below.
- **area**: A pointer to a communications area in your process that the *handler* can assume is never paged out, or NULL if you don’t want a communications area.
- **size**: The size of the communications area.
- **flags**: Flags that specify how you want to attach the interrupt handler. For more information, see “Flags,” below.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `InterruptAttach()` and `InterruptAttach_r()` kernel calls attach the interrupt function handler to the hardware interrupt specified by `intr`. They automatically enable (i.e. unmask) the interrupt level.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Before calling either of these functions, the thread must request I/O privileges by calling:

```c
ThreadCtl(_NTO_TCTL_IO, 0);
```

If the thread doesn’t do this, the attachment fails with an error code of `EPERM`.

Interrupt vector numbers

The interrupt values for `intr` are logical interrupt vector numbers grouped into related “interrupt classes” that generally correspond to a particular interrupt line on the CPU. The following interrupt classes are present on all QNX Neutrino systems:

- `_NTO_INTR_CLASS_EXTERNAL`
  - Normal external interrupts (such as the ones generated by the `INTR` pin on x86 CPUs).
- `_NTO_INTR_CLASS_SYNTHETIC`
  - Synthetic, kernel-generated interrupts.

- `_NTO_INTR_SPARE` is usually the only `_NTO_INTR_CLASS_SYNTHETIC` interrupt you’ll use;
- `_NTO_INTR_SPARE` is guaranteed not to match any valid logical interrupt vector number.

There can be additional interrupt classes defined for specific CPUs or embedded systems. For the interrupt assignments for specific boards, see the sample build files in `$QNXTARGET/$PROCESSOR/boot/build`. 

September 10, 2007
Interrupts and startup code

The mapping of logical interrupt vector numbers is completely dependent on the implementor of the startup code.

Device drivers must:

- Let the user specify an interrupt number on the command line; don’t use a hard-coded value. Eventually, the configuration manager will provide interrupt numbers for the device drivers.

- Store interrupt numbers in an `unsigned int` variable; don’t assume an interrupt number fits into a byte.

Typical x86 Interrupt vector numbers

The following list contains typical interrupt assignments for the 16 hardware interrupts on an x86-based PC using `startup-bios`:

<table>
<thead>
<tr>
<th>Interrupt <code>intr</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A clock that runs at the resolution set by <code>ClockPeriod()</code></td>
</tr>
<tr>
<td>1</td>
<td>Keyboard</td>
</tr>
<tr>
<td>2</td>
<td>Slave 8259 — you can’t attach to this interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>Com2</td>
</tr>
<tr>
<td>4</td>
<td>Com1</td>
</tr>
<tr>
<td>5</td>
<td>Net card / sound card / other</td>
</tr>
<tr>
<td>6</td>
<td>Floppy</td>
</tr>
<tr>
<td>7</td>
<td>Parallel printer / sound card / other</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Remapped interrupt 2</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

*continued...*
**Interrupt intr**  **Description**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Co-processor</td>
</tr>
<tr>
<td>14</td>
<td>Primary disk controller</td>
</tr>
<tr>
<td>15</td>
<td>Secondary disk controller</td>
</tr>
</tbody>
</table>

The interrupt assignments are different for other boards.

**Interrupt handler function**

The function to call is specified by the `handler` argument. This function runs in the environment of your process. If a pager is running that swaps pages out of memory, it’s possible for your `handler` to reference a variable in the process address space that isn’t present. This results in a kernel shutdown.

The `area` and `size` arguments define a communications area in your process that the `handler` can assume is never paged out. This typically is a structure containing buffers and information needed by the `handler` and the process when it runs. In a paging system, lock the memory pointed to by `area` by calling `mlock()` before attaching the `handler`. In a nonpaging system, you can omit the call to `mlock()` (but you should still call it for compatibility with future versions of the OS).

The `area` argument can be NULL to indicate no communications area. If `area` is NULL, `size` should be 0.

The `handler` function’s prototype is:

```c
const struct sigevent* handler( void* area, int id );
```

Where `area` is a pointer to the `area` specified by the call to `InterruptAttach()`, and `id` is the ID returned by `InterruptAttach()`.
Follow the following guidelines when writing your handler:

- A temporary interrupt stack of limited depth is provided at interrupt time, so avoid placing large arrays or structures on the stack frame of the handler. It’s safe to assume that about 200 bytes of stack are available.

- The interrupt handler runs asynchronously with the threads in the process. Any variables modified by the handler should be declared with the `volatile` keyword and modified with interrupts disabled or using the `atomic*()` functions in any thread and ISR.

- The interrupt handler should be kept as short as possible. If a significant amount of work needs to be done, the handler should deliver an event to awaken a thread to do the work.

- The handler can’t call library routines that contain kernel calls except for `InterruptDisable()`, `InterruptEnable()`, `InterruptLock()`, `InterruptMask()`, `InterruptUnlock()`, and `InterruptUnmask()`.

  The handler can call `TraceEvent()`, but not all modes are valid.

The return value of the `handler` function should be NULL or a pointer to a valid `sigevent` structure that the kernel delivers. These events are defined in `<signal.h>`.

Consider the following when choosing the event type:

- Message-driven processes that block in a receive loop using `MsgReceivev()` should consider using `SIGEV_PULSE` to trigger a pulse.

- Threads that block at a particular point in their code and don’t go back to a common receive point should consider using `SIGEV_INTR` as the event notification type and `InterruptWait()` as the blocking call.
The thread that calls `InterruptWait()` must be the one that called `InterruptAttach()`.

- Using SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, SIGEV_SIGNAL_THREAD, or SIGEV_THREAD is discouraged. It’s less efficient than the other mechanisms for interrupt event delivery.

**Flags**

The `flags` argument is a bitwise OR of the following values, or 0:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_INTR_FLAGS_END</td>
<td>Put the new handler at the end of the list of existing handlers (for shared interrupts) instead of the start.</td>
</tr>
<tr>
<td>_NTO_INTR_FLAGS_PROCESS</td>
<td>Associate the handler with the process instead of the attaching thread.</td>
</tr>
<tr>
<td>_NTO_INTR_FLAGS_TRK_MSK</td>
<td>Track calls to <code>InterruptMask()</code> and <code>InterruptUnmask()</code> to make detaching the interrupt handler safer.</td>
</tr>
</tbody>
</table>

The interrupt structure allows hardware interrupts to be shared. For example, if two processes take over the same physical interrupt, both handlers are invoked consecutively. When a handler attaches, it’s placed in front of any existing handlers for that interrupt and is called first. You can change this behavior by setting the `_NTO_INTR_FLAGS_END` flag in the `flags` argument. This adds the...
handler at the end of any existing handlers. Although the Neutrino microkernel allows full interrupt sharing, your hardware might not. For example, the ISA bus doesn’t allow interrupt sharing, while the PCI bus does.

Processor interrupts are enabled during the execution of the handler. Don’t attempt to talk to the interrupt controller chip. The operating system issues the end-of-interrupt command to the chip after processing all handlers at a given level.

The first process to attach to an interrupt unmasks the interrupt. When the last process detaches from an interrupt, the system masks it.

If the thread that attached the interrupt handler terminates without detaching the handler, the kernel does it automatically.

## _NTO_INTR_FLAGS_PROCESS

Adding _NTO_INTR_FLAGS_PROCESS to flags associates the interrupt handler with the process instead of the attaching thread. The interrupt handler is removed when the process exits, instead of when the attaching thread exits.

## _NTO_INTR_FLAGS_TRK_MSK

The _NTO_INTR_FLAGS_TRK_MSK flag and the id argument to InterruptMask() and InterruptUnmask() let the kernel track the number of times a particular interrupt handler or event has been masked. Then, when an application detaches from the interrupt, the kernel can perform the proper number of unmasks to ensure that the interrupt functions normally. This is important for shared interrupt levels.

You should always set _NTO_INTR_FLAGS_TRK_MSK.

### Blocking states

This call doesn’t block.
Returns:

The only difference between these functions is the way they indicate errors:

InterruptAttach()

An interrupt function ID. If an error occurs, -1 is returned and errno is set.

InterruptAttach_r()

An interrupt function ID. This function does NOT set errno. If an error occurs, the negative of a value from the Errors section is returned.

Use the function ID with the InterruptDetach() function to detach this interrupt handler.

Errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>All kernel interrupt entries are in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the buffers provided.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value of intr isn’t a valid interrupt number.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The process doesn’t have I/O privileges.</td>
</tr>
</tbody>
</table>

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
InterruptAttach(), InterruptAttach_r()

Caveats:

If you’re writing a resource manager and using the resmgr_*( ) functions with multiple threads, a thread that attaches to an interrupt must use _NTO_INTR_FLAGS_PROCESS in the flags argument when calling InterruptAttach().

If your interrupt handler isn’t SMP-safe, you must lock it to one processor using:

ThreadCtl(_NTO_TCTL_RUNMASK, ...);

See also:

atomic_add(), atomic_clr(), atomic_set(), atomic_sub(),
atomic_toggle(), InterruptAttachEvent(), InterruptDetach(),
InterruptDisable(), InterruptEnable(), InterruptLock(),
InterruptMask(), InterruptUnlock(), InterruptUnmask(),
InterruptWait(), mlock(), sigevent, ThreadCtl(), TraceEvent()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptAttachEvent(), InterruptAttachEvent_r()

Synopsis:

```c
#include <sys/neutrino.h>

int InterruptAttachEvent(
    int intr,
    const struct sigevent* event,
    unsigned flags);

int InterruptAttachEvent_r(
    int intr,
    const struct sigevent* event,
    unsigned flags);
```

Arguments:

- `intr` The interrupt vector number that you want to attach an event to; for more information, see “Interrupt vector numbers” in the documentation for InterruptAttach().

- `event` A pointer to the sigevent structure that you want to be delivered when this interrupt occurs.

- `flags` Flags that specify how you want to attach the interrupt handler. For more information, see “Flags,” below.

Library:

- libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The InterruptAttachEvent() and InterruptAttachEvent_r() kernel calls attach the given event to the hardware interrupt specified by `intr`. They automatically enable (i.e unmask) the interrupt level.

The InterruptAttachEvent() and InterruptAttachEvent_r() functions are identical except in the way they indicate errors. See the Returns section for details.
InterruptAttachEvent(), InterruptAttachEvent_r() ©
2007, QNX Software Systems GmbH & Co. KG.

Before calling either of these functions, the thread must request I/O privileges by calling:

ThreadCtl( _NTO_TCTL_IO, 0 );

If the thread doesn’t do this, it might SIGSEGV when it calls InterruptAttachEvent() or InterruptAttachEvent_r().

To prevent infinite interrupt recursion, the kernel automatically does an InterruptMask() for intr when delivering the event. After the interrupt-handling thread has dealt with the event, it must call InterruptUnmask() to reenable the interrupt.

Consider the following when choosing an event type:

- Message-driven processes that block in a receive loop using MsgReceivev() should consider using SIGEV_PULSE to trigger a channel.

- Threads that block at a particular point in their code and don’t go back to a common receive point, should consider using SIGEV_INTR as the event notification type and InterruptWait() as the blocking call.

The thread that calls InterruptWait() must be the one that called InterruptAttachEvent().

- Using SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, or SIGEV_SIGNALTHREAD is discouraged. It is less efficient than the other mechanisms for interrupt event delivery.

Flags

The flags argument is a bitwise OR of the following values, or 0:
InterruptAttachEvent(), InterruptAttachEvent_r()

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_INTR_FLAGS_END</td>
<td>Put the new event at the end of the list of existing events instead of the start.</td>
</tr>
<tr>
<td>_NTO_INTR_FLAGS_PROCESS</td>
<td>Associate the event with the process instead of the attaching thread.</td>
</tr>
<tr>
<td>_NTO_INTR_FLAGS_TRK_MSK</td>
<td>Track calls to InterruptMask() and InterruptUnmask() to make detaching the interrupt handler safer.</td>
</tr>
</tbody>
</table>

_NTO_INTR_FLAGS_END

The interrupt structure allows hardware interrupts to be shared. For example if two processes call InterruptAttachEvent() for the same physical interrupt, both events are sent consecutively. When an event attaches, it’s placed in front of any existing events for that interrupt and is delivered first. You can change this behavior by setting the _NTO_INTR_FLAGS_END flag in the flags argument. This adds the event at the end of any existing events.

_NTO_INTR_FLAGS_PROCESS

Adding _NTO_INTR_FLAGS_PROCESS to flags associates the interrupt event with the process instead of the attaching thread. The interrupt event is removed when the process exits, instead of when the attaching thread exits.

The kernel automatically attempts to set the _NTO_INTR_FLAGS_PROCESS flag if the event is directed at the process in general (for SIGEV_SIGNAL, SIGEV_SIGNAL_CODE, and SIGEV_PULSE events).
InterruptAttachEvent(), InterruptAttachEvent_r() ©

2007, QNX Software Systems GmbH & Co. KG.

_NTO_INTR_FLAGS_TRK_MSK

The _NTO_INTR_FLAGS_TRK_MSK flag and the id argument to InterruptMask() and InterruptUnmask() let the kernel track the number of times a particular interrupt handler or event has been masked. Then, when an application detaches from the interrupt, the kernel can perform the proper number of unmask to ensure that the interrupt functions normally. This is important for shared interrupt levels.

You should always set _NTO_INTR_FLAGS_TRK_MSK.

Advantages & disadvantages

InterruptAttachEvent() has several advantages over InterruptAttach():

- Less work is done at interrupt time (you avoid the context switch necessary to map in an interrupt handler).
- Interrupt handling code runs at the thread’s priority, which lets you specify the priority of the interrupt handling.
- You can use process-level debugging on your interrupt handler code.

There are also some disadvantages:

- There might be a delay before the interrupt handling code runs (until the thread is scheduled to run).
- For multiple devices sharing an event, the amount of time spent with the interrupt masked increases.

You can freely mix calls to InterruptAttach() and InterruptAttachEvent() for a particular interrupt.
InterruptAttachEvent(), InterruptAttachEvent_r()

Blocking states

This call doesn’t block.

Returns:

The only difference between these functions is the way they indicate errors:

InterruptAttachEvent()

An interrupt function ID. If an error occurs, -1 is returned and errno is set.

InterruptAttachEvent_r()

An interrupt function ID. This function does NOT set errno. If an error occurs, the negative of a value from the Errors section is returned.

Use the ID with InterruptDetach() to detach this interrupt event.

Errors:

EAGAIN All kernel interrupt entries are in use.
EFAULT A fault occurred when the kernel tried to access the buffers provided.
EINVAL The value of intr isn’t a valid interrupt number.
EPERM The process doesn’t have superuser capabilities.

Classification:

QNX Neutrino

Safety

Cancellation point No

continued . . .
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*InterruptAttach(), InterruptDetach(), InterruptLock(), InterruptMask(), InterruptUnlock(), InterruptUnmask(), InterruptWait(), sigevent*

Writing an Interrupt Handler chapter of the Neutrino *Programmer’s Guide*
Synopsis:

```c
#include <sys/neutrino.h>

int InterruptDetach( int id );
int InterruptDetach_r( int id );
```

Arguments:

- `id` The value returned by `InterruptAttach()`.
  `InterruptAttachEvent()`, or `InterruptHookIdle()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls detach the interrupt handler specified by the `id` argument. If, after detaching, no thread is attached to the interrupt then the interrupt is masked off.

The `InterruptDetach()` and `InterruptDetach_r()` functions are identical except in the way they indicate errors. See the Returns section for details.

Before calling either of these functions, the thread must request I/O privileges by calling:

```c
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptDetach()` or `InterruptDetach_r()`.
**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

*InterruptDetach()*

If an error occurs, -1 is returned and `errno` is set. Any other value returned indicates success.

*InterruptDetach_r()*

EOK is returned on success. This function does **NOT** set `errno`. If an error occurs, any value in the Errors section may be returned.

**Errors:**

- **EINVAL** The value of `id` doesn’t exist for this process.
- **EPERM** The process doesn’t have superuser capabilities.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

InterruptAttach(), InterruptAttachEvent(), InterruptHookIdle(), InterruptUnlock()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptDisable()  © 2007, QNX Software Systems GmbH & Co. KG.

Disable hardware interrupts

Synopsis:

```c
#include <sys/neutrino.h>

void InterruptDisable( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptDisable()` function disables all hardware interrupts. You can call it from a thread or from an interrupt handler. Before calling this function, the thread must request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

Any kernel call results in the re-enabling of interrupts, and many library routines are built on kernel calls. Masked interrupts are not affected.

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptDisable()`.

Reenable the interrupts by calling `InterruptEnable()`.

**CAUTION:** Since this function disables all hardware interrupts, take care to reenable them as quickly as possible. Failure to do so may result in increased interrupt latency and nonrealtime performance.

Use `InterruptDisable()` instead of an inline `cli` to ensure hardware portability with non-x86 CPUs.
Use `InterruptLock()` and `InterruptUnlock()` instead of `InterruptDisable()` and `InterruptEnable()`. The `InterruptLock()` and `InterruptUnlock()` functions perform the intended function on SMP hardware, and allow your interrupt thread to run on any processor in the system.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`InterruptEnable()`, `InterruptLock()`, `InterruptMask()`, `InterruptUnlock()`, `InterruptUnmask()`, `ThreadCtl()`

Writing an Interrupt Handler chapter of the Neutrino *Programmer’s Guide*
**InterruptEnable()**

*Enable hardware interrupts*

**Synopsis:**
```
#include <sys/neutrino.h>

void InterruptEnable( void );
```

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `InterruptEnable()` function enables all hardware interrupts. You can call it from a thread or from an interrupt handler. Before calling this function, the thread must request I/O privileges by calling:
```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptEnable()`.

You should call this function as quickly as possible after calling `InterruptDisable()`.

---

Use `InterruptLock()` and `InterruptUnlock()` instead of `InterruptDisable()` and `InterruptEnable()`. The `InterruptLock()` and `InterruptUnlock()` functions perform the intended function on SMP hardware, and allow your interrupt thread to run on any processor in the system.

**Classification:**
QNX Neutrino
InterruptEnable()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

InterruptDisable(), InterruptLock(), InterruptMask(), InterruptUnlock(), InterruptUnmask(), ThreadCtl()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
Attach an “idle” interrupt handler

Synopsis:

```c
#include <sys/neutrino.h>

int InterruptHookIdle(
    void (*handler)(uint64_t *, struct qtime_entry *),
    unsigned flags);
```

Arguments:

- `handler` A pointer to the handler function; see below.
- `flags` Flags that specify how you want to attach the interrupt handler. For more information, see “Flags,” below.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptHookIdle()` kernel call attaches the specified interrupt handler to the “idle” interrupt, which is called when the system is idle. This is typically used to implement power management features.

The arguments to the `handler` functions are:

- `uint64_t*` A pointer to the time, in nanoseconds, when the next timer will expire.
- `struct qtime_entry*` A pointer to the section of the system page with the time information, including the current time of day.

The simplest idle handler consists of a `halt` instruction.
Flags

The flags argument is a bitwise OR of the following values, or 0:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_INTR_FLAGS_END</td>
<td>Put the new handler at the end of the list of existing handlers (for shared interrupts) instead of the start.</td>
</tr>
<tr>
<td>_NTO_INTR_FLAGS_PROCESS</td>
<td>Associate the handler with the process instead of the attaching thread.</td>
</tr>
<tr>
<td>_NTO_INTR_FLAGS_TRK_MSK</td>
<td>Track calls to InterruptMask() and InterruptUnmask() to make detaching the interrupt handler safer.</td>
</tr>
</tbody>
</table>

_NTO_INTR_FLAGS_END

The interrupt structure allows hardware interrupts to be shared. For example, if two processes take over the same physical interrupt, both handlers are invoked consecutively. When a handler attaches, it’s placed in front of any existing handlers for that interrupt and is called first. You can change this behavior by setting the _NTO_INTR_FLAGS_END flag in the flags argument. This adds the handler at the end of any existing handlers.

Processor interrupts are enabled during the execution of the handler. Don’t attempt to talk to the interrupt controller chip. The end of interrupt command is issued to the chip by the operating system after processing all handlers at a given level.

The first process to attach to an interrupt un masks the interrupt. When the last process detaches from an interrupt, the system masks it.

If the thread that attached the interrupt handler terminates without detaching the handler, the kernel does it automatically.
Adding _NTO_INTR_FLAGS_PROCESS to flags associates the interrupt handler with the process instead of the attaching thread. The interrupt handler is removed when the process exits, instead of when the attaching thread exits.

The _NTO_INTR_FLAGS_TRK_MSK flag and the id argument to InterruptMask() and InterruptUnmask() let the kernel track the number of times a particular interrupt handler or event has been masked. Then, when an application detaches from the interrupt, the kernel can perform the proper number of unmasks to ensure that the interrupt functions normally. This is important for shared interrupt values.

Blocking states
This call doesn’t block.

Returns:
An interrupt function ID, or -1 if an error occurs (errno is set).

Use the returned value with the InterruptDetach() function to detach this interrupt handler.

Errors:
EAGAIN All kernel interrupt entries are in use.
EPERM The process doesn’t have superuser capabilities.

Classification:
QNX Neutrino
InterruptHookIdle()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

InterruptAttach(), InterruptAttachEvent(), InterruptDetach(), InterruptHookTrace()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptHookTrace() © 2007, QNX Software Systems GmbH & Co. KG.

Attach the pseudo interrupt handler that the instrumented module uses

Synopsis:

```c
#include <sys/neutrino.h>

int InterruptHookTrace(
    const struct sigevent * (* handler)(int),
    unsigned flags);
```

Arguments:

- `handler`: A pointer to the handler function.
- `flags`: Flags that specify how you want to attach the interrupt handler.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptHookTrace()` kernel call attaches the pseudo interrupt handler `handle` that the instrumented module uses.

This function requires the instrumented kernel. For more information, see the documentation for the System Analysis Toolkit (SAT).

Returns:

An interrupt function ID, or -1 if an error occurs (`errno` is set).

Errors:

- `EAGAIN`: All kernel interrupt entries are in use.
- `EFAULT`: A fault occurred when the kernel tried to access the buffers provided.
InterruptHookTrace()

EPERM       The process doesn’t have superuser capabilities.
ENOTSUP     The kernel is not instrumented.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

InterruptAttach(), TraceEvent()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
**InterruptLock()**

Guard a critical section in an interrupt handler

**Synopsis:**

```
#include <sys/neutrino.h>

void InterruptLock( intrspin_t* spinlock );
```

**Arguments:**

- **spinlock**: The spinlock (a variable shared between the interrupt handler and a thread) to use.

  If `spinlock` isn’t a `static` variable, you must initialize it by calling:

  ```
  memset( spinlock, 0, sizeof( *spinlock ) );
  ```

  before using it with `InterruptLock()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `InterruptLock()` function guards a critical section by locking the specified `spinlock`. You can call this function from a thread or from an interrupt handler. Before calling this function, the thread must request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptLock()`.

This function tries to acquire the `spinlock` (a variable shared between the interrupt handler and a thread) while interrupts are disabled. The code spins in a tight loop until the lock is acquired. It’s important to release the lock as soon as possible. Typically, this is a few lines of code without any loops:
InterruptLock( &spinner );
/* ... critical section */
InterruptUnlock( &spinner );

InterruptLock() solves a common need in many realtime systems to protect access to shared data structures between an interrupt handler and the thread that owns the handler. The traditional POSIX primitives used between threads aren’t available for use by an interrupt handler.

The InterruptLock() and InterruptUnlock() functions work on single-processor or multiprocessor machines.

Any kernel call results in the re-enabling of interrupts, and many library routines are built on kernel calls. Masked interrupts are not affected.

Classification:
QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
InterruptDisable(), InterruptEnable(), InterruptMask(), InterruptUnlock(), InterruptUnmask(), ThreadCtl()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptMask()  
Disable a hardware interrupt

Synopsis:

```c
#include <sys/neutrino.h>

int InterruptMask( int intr,
                   int id );
```

Arguments:

- `intr` The interrupt you want to mask.
- `id` The value returned by `InterruptAttach()` or `InterruptAttachEvent()`, or -1 if you don’t want the kernel to track interrupt maskings and unmaskings for each handler.

The `id` is ignored unless you use the `_NTO_INTR_FLAGS_TRK_MSK` flag when you attach the handler.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptMask()` kernel call disables the hardware interrupt specified by `intr` for the handler specified by `id`. You can call this function from a thread or from an interrupt handler. Before calling this function, the thread must request I/O privileges by calling:

```c
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptMask()`.

Reenable the interrupt by calling `InterruptUnmask()`.

The kernel automatically enables an interrupt when the first handler attaches to it using `InterruptAttach()` and disables it when the last handler detaches.
InterruptMask()

This call is often used when a device presents a level-sensitive interrupt to the system that can’t be easily cleared in the interrupt handler. Since the interrupt is level-sensitive, you can’t exit the handler with the interrupt line active and unmasked. InterruptMask() lets you mask the interrupt in the handler and schedule a thread to do the real work of communicating with the device to clear the source. Once cleared, the thread should call InterruptUnmask() to reenable this interrupt.

To disable all hardware interrupts, use the InterruptLock() function.

To ensure hardware portability, use InterruptMask() instead of writing code that talks directly to the interrupt controller.

Calls to InterruptMask() are nested; the interrupt isn’t unmasked until InterruptUnmask() has been called once for every call to InterruptMask().

Returns:

The current mask level count for success; or -1 if an error occurs (errno is set).

Errors:

EINVAL   The value of intr isn’t a supported hardware interrupt.

Classification:

QNX Neutrino

Safety

Cancellation point   No
Interrupt handler    Yes
Signal handler       Yes

continued…
InterruptMask()

Safety
Thread: Yes

See also:
InterruptAttach(), InterruptDisable(), InterruptEnable(),
InterruptLock(), InterruptUnlock(), InterruptUnmask(), ThreadCtl()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptUnlock()  
Release a critical section in an interrupt handler

Synopsis:

```c
#include <sys/neutrino.h>

void InterruptUnlock( intrspin_t* spinlock);
```

Arguments:

`spinlock`  
The spinlock (a variable shared between the interrupt handler and a thread) used in a call to `InterruptLock()` to lock the handler.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptUnlock()` function releases a critical section by unlocking the specified `spinlock`, reenabling interrupts. You can call this function from a thread or from an interrupt handler.

Before calling this function, the thread must request I/O privileges by calling:

```c
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptUnlock()`.

Classification:

QNX Neutrino

Safety

Cancellation point   No

`continued...`
Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

InterruptDisable(), InterruptEnable(), InterruptLock(), InterruptMask(), InterruptUnmask(), ThreadCtl()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
Synopsis:

```c
#include <sys/neutrino.h>

int InterruptUnmask( int intr,
                      int id );
```

Arguments:

- `intr` The interrupt you want to unmask.
- `id` The value returned by `InterruptAttach()` or `InterruptAttachEvent()`, or -1 if you don’t want the kernel to track interrupt maskings and unmaskings for each handler.

The `id` is ignored unless you use the `_NTO_INTR_FLAGS_TRK_MSK` flag when you attach the handler.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `InterruptUnmask()` kernel call enables the hardware interrupt specified by `intr` for the interrupt handler specified by `intr` for the handler specified by `id` when the mask count reaches zero. You can call this function from a thread or from an interrupt handler. Before calling this function, the thread must request I/O privileges by calling:

```c
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptUnmask()`.

Calls to `InterruptMask()` are nested; the interrupt isn’t unmasked until `InterruptUnmask()` has been called once for every call to `InterruptMask()`.
InterruptUnmask()

Returns:
The current mask count, or -1 if an error occurs (errno is set).

Errors:
EINVAL Not a supported hardware interrupt intr.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
InterruptAttach(), InterruptDisable(), InterruptEnable(),
InterruptLock(), InterruptMask(), InterruptUnlock() ThreadCtl()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptWait(), InterruptWait_r()

Wait for a hardware interrupt

Synopsis:

```c
#include <sys/neutrino.h>

int InterruptWait( int flags,
                   const uint64_t *timeout );

int InterruptWait_r( int flags,
                     const uint64_t *timeout );
```

Arguments:

- **flags**  This should currently be 0.
- **timeout**  This should currently be NULL. This may change in future versions.

Use TimerTimeout() to achieve a timeout.

Library:

- libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

These kernel calls wait for a hardware interrupt. The calling thread should have attached a handler to the interrupt, by calling InterruptAttach() or InterruptAttachEvent(). The call to InterruptWait() or InterruptWait_r() blocks waiting for an interrupt handler to return an event with notification type SIGEV_INTR (i.e. a hardware interrupt).

The InterruptWait() and InterruptWait_r() functions are identical except in the way they indicate errors. See the Returns section for details.

If the notification event occurs before InterruptWait() is called, a pending flag is set. When InterruptWait() is called, the flag is checked; if set, it’s cleared and the call immediately returns with success.
**InterruptWait(), InterruptWait_r()**

*© 2007, QNX Software Systems GmbH & Co. KG.*

### Blocking states

STATE_INTR  The thread is waiting for an interrupt handler to return a SIGEV_INTR event.

### Returns:

The only difference between these functions is the way they indicate errors:

**InterruptWait()**

If an error occurs, -1 is returned and *errno* is set. Any other value returned indicates success.

**InterruptWait_r()**

EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

### Errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINTR</td>
<td>The call was interrupted by a signal.</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>The reserved arguments aren’t NULL.</td>
</tr>
</tbody>
</table>
| ETIMEDOUT | A kernel timeout unblocked the call. See *TimerTimeout()*.

### Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
</tbody>
</table>

*continued...*
Safety

| Thread | Yes |

See also:

InterruptAttach(), InterruptAttachEvent(), TimerTimeout()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
intr_v86()

Execute a real-mode software interrupt

Synopsis:

```c
#include <x86/v86.h>

int __intr_v86( int swi,
                struct _v86reg* regs,
                void* data,
                int datasize );
```

Arguments:

- `swi` The software interrupt that you want to execute.
- `regs` A pointer to a _v86reg structure that specifies the values you want to use for the registers on entry to real mode; see below.
- `data` A pointer to the data that you want to copy into memory; see below.
- `datasize` The size of the data, in bytes.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The intr_v86() function executes the real-mode software interrupt specified by `swi` in virtual 8086 mode. This allows access to the ROM BIOS functions that are designed to run in 16-bit real mode. Two common examples are:
<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int 10h</td>
<td>Video BIOS</td>
</tr>
<tr>
<td>int 1ah</td>
<td>PCI</td>
</tr>
</tbody>
</table>

BIOS calls (such as int 13h, disk I/O) that require hardware interrupts to be directed at their code aren’t supported.

Upon entry to real mode, the registers are loaded from regs. The segment registers and any pointers should address a 2K communication area located at offset 0:800h in real memory. The buffer data of length datasize is copied to this area just before real mode is entered and copied back when the call completes. At this point regs is also updated to contain the values of the real-mode registers.

You should set the DS, ES, FS and GS segment registers to 0. The values in the CS:IP, and SS:SP registers are ignored and are set by the kernel. The stack provided is about 500 bytes in size.

The layout of real mode memory is described by the structure _v86_memory in `<x86/v86.h>`.

When a thread enters virtual 8086 mode, all threads in the system continue to be scheduled based upon their priority, including the calling thread. While in virtual 8086 mode, full access to IO ports and interrupt enable and disable are allowed. Only one thread may enter virtual 8086 mode at a time.

This function fails if the calling process doesn’t have an effective user ID of root (euid 0).

**Returns:**

0       Success.

-1      An error occurred; errno is set.
Errors:

EPERM The calling thread didn’t have an effective user ID of root.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <errno.h>
#include <x86/v86.h>

struct _v86reg reg;

int main( void ) {
    char buf[4];

    /* Equipment call */
    printf("int 12\n");
    memset(&reg, 0, sizeof(reg));
    _intr_v86(0x12, &reg, NULL, 0);
    printreg();
    sleep(5);

    /* Enter 40 column text mode */
    printf("int 10 ah=00h al=00h\n");
    memset(&reg, 0, sizeof(reg));
    _intr_v86(0x10, &reg, NULL, 0);
    printreg();
    sleep(5);

    /* Enter 80 column text mode */
    printf("int 10 ah=00h al=02h\n");
    memset(&reg, 0, sizeof(reg));
    reg.eax = 2;
    _intr_v86(0x10, &reg, NULL, 0);
    printreg();
    sleep(5);

    /* Write a string from memory */
    printf("int 10 ah=13h al=00h\n");
    strcpy(buf, "Hi!");
    memset(&reg, 0, sizeof(reg));
    reg.eax = 0x1300;
    reg.es = 0;
    reg.ebp = offsetof(struct _v86_memory, userdata);
    reg.ecx = strlen(buf);
    reg.edx = 0;
    reg.ebx = 0x0007;
```
_intr_v86(0x10, &reg, buf, strlen(buf));
printreg();
sleep(5);
return EXIT_SUCCESS;
}

printreg() {
    printf("eax=%-8x ebx=%-8x ecx=%-8x edx=%-8x\n", reg.eax, reg.ebx, reg.ecx, reg.edx);
    printf("esi=%-8x edi=%-8x ebp=%-8x esp=%-8x\n", reg.esi, reg.edi, reg.ebp, reg.esp);
    printf(" ds=%-8x  es=%-8x  fs=%-8x  gs=%-8x\n", reg.ds, reg.es, reg.fs, reg.gs);
    printf("efl=%-8x\n", reg.efl);
}

Classification:
QNX Neutrino (x86 only)

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Structure of a resource manager’s connect message

Synopsis:
```
struct _io_connect {
    uint16_t type;
    uint16_t subtype;
    uint32_t file_type;
    uint16_t reply_max;
    uint16_t entry_max;
    uint32_t key;
    uint32_t handle;
    uint32_t ioflag;
    uint32_t mode;
    uint16_t sflag;
    uint16_t access;
    uint16_t zero;
    uint16_t path_len;
    uint8_t eflag;
    uint8_t extra_type;
    uint16_t extra_len;
    char path[1];
};
```

Description:
The `_io_connect` structure is used to describe a connect message that a resource manager receives and sends.

The members include:

- **type**
  - _IO_CONNECT

- **subtype**
  - The type of connection that the message concerns; one of:
    - _IO_CONNECT_COMBINE — combine with an I/O message.
    - _IO_CONNECT_COMBINE_CLOSE — combine with I/O message and always close.
    - _IO_CONNECT_OPEN
    - _IO_CONNECT_UNLINK
• _IO_CONNECT_RENAME
• _IO_CONNECT_MKNOD
• _IO_CONNECT_READLINK
• _IO_CONNECT_LINK
• _IO_CONNECT_RSVD_UNBLOCK — place holder in the jump table.
• _IO_CONNECT_MOUNT

_file_type_ The file type; one of the following (defined in `<sys/ftype.h>`):

• _FTYPE_ANY — the path name can be anything.
• _FTYPE_LINK — reserved for the Process Manager.
• _FTYPE_MOUNT — receive mount requests on the path (path must be NULL).
• _FTYPE_MQUEUE — reserved for a message-queue manager.
• _FTYPE_PIPE — reserved for a pipe manager.
• _FTYPE_SEM — reserved for a semaphore manager.
• _FTYPE_SHMEM — reserved for a shared memory object.
• _FTYPE_SOCKET — reserved for a socket manager.
• _FTYPE_SYMLINK — reserved for the Process Manager.

_reply_max_ The maximum length of the reply message.

_entry_max_ The maximum number of _io_connect_entry structures that the resource manager is willing to accept. If a path could reference more than one resource manager, it returns a list of _io_connect_entry structures referring to the overlapping resource managers.
key  
Reserved.

handle  
The handle returned by resmgr_attach().

ioflag  
The bottom two bits are modified from traditional Unix values to more useful bit flags:

- O_RDONLY (0) is converted to _IO_FLAG_RD (0x01)
- O_WRONLY (1) is converted to _IO_FLAG_WR (0x02)
- O_RDWR (2) is converted to _IO_FLAG_RD | _IO_FLAG_WR (0x03)

Note that this translation can be performed without overlapping into other O_* flags.

Testing is done using:

- ioflag & _IO_FLAG_RD; for read permission.
- ioflag & _IO_FLAG_WR; for write permission.
- If open for reading and writing — both bits are set.

The remaining values of ioflag are outside this range and are not modified. These values are:

- O_APPEND — if set, the file offset is set to the end of the file prior to each write.
- O_CREAT — create the file.
- O_DSYNC — if set, this flag affects subsequent I/O calls; each call to write() waits until all data is successfully transferred to the storage device such that it’s readable on any subsequent open of the file (even one that follows a system failure) in the absence of a failure of the physical storage medium. If the physical storage medium implements a non-write-through cache, then a system failure may be interpreted as a failure of the physical storage medium, and data may not be
 readable even if this flag is set and the write() indicates that it succeeded.

- O_EXCL — if you set both O_EXCL and O_CREAT, open() fails if the file exists. The check for the existence of the file and the creation of the file if it doesn’t exist are atomic; no other process that’s attempting the same operation with the same filename at the same time will succeed. Specifying O_EXCL without O_CREAT has no effect.

- O_LARGEFILE — allow the file offset to be 64 bits long.

- O_NOCTTY — if set, and path identifies a terminal device, the open() function doesn’t cause the terminal device to become the controlling terminal for the process.

- O_NONBLOCK — don’t block.

- O_REALIDS — use the real uid/gid for permissions checking.

- O_RSYNC — read I/O operations on the file descriptor complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags.

- O_SYNC — if set, this flag affects subsequent I/O calls; each call to read() or write() is complete only when both the data has been successfully transferred (either read or written) and all file system information relevant to that I/O operation (including that required to retrieve said data) is successfully transferred, including file update and/or access times, and so on. See the discussion of a successful data transfer in O_DSYNC, above.

- O_TRUNC — if the file exists and is a regular file, and the file is successfully opened O_WRONLY or O_RDWR, the file length is truncated to zero and the mode and owner are left unchanged. O_TRUNC has no effect on FIFO or block or character special
files or directories. Using O_TRUNC with O_RDONLY has no effect.  

Contaıns the type and access permissions of the file.  
The type is one of:  

- S_IFBLK — block special.  
- S_IFCHR — character special.  
- S_IFDIR — directory.  
- S_IFIFO — FIFO special.  
- S_IFLNK — symbolic link.  
- S_IFMT — type of file.  
- S_IFNAM — special named file.  
- S_IFREG — regular.  
- S_IFSOCK — socket.  

The permissions are a combination of:  

<table>
<thead>
<tr>
<th>Owner</th>
<th>Group</th>
<th>Others</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IRUSR</td>
<td>S_IRGRP</td>
<td>S_IROTH</td>
<td>Read</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>S_IRWXG</td>
<td>S_IRWXR</td>
<td>Read, write, execute/search. A bitwise inclusive OR of the other three constants (S_IRWXU is OR of IRUSR, S_IWSUR and S_IXUSR.)</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>S_IWGRP</td>
<td>S_IWOTH</td>
<td>Write</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>S_IXGRP</td>
<td>S_IXOTH</td>
<td>Execute/search</td>
</tr>
</tbody>
</table>

The following bits define miscellaneous permissions used by other implementations:
### Bit Equivalent

<table>
<thead>
<tr>
<th>Bit</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IEXEC</td>
<td>S_IXUSR</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>S_IRUSR</td>
</tr>
<tr>
<td>S_IWRITE</td>
<td>S_IWUSR</td>
</tr>
</tbody>
</table>

#### sflag
How the client wants the file to be shared; a combination of the following bits:

- **SH_COMPAT** — set compatibility mode.
- **SH_DENYRW** — prevent read or write access to the file.
- **SH_DENYWR** — prevent write access to the file.
- **SH_DENYRD** — prevent read access to the file.
- **SH_DENYNO** — permit both read and write access to the file.

#### access
Contains a combination of **_IO_FLAG_RD** and/or **_IO_FLAG_WR** bits, which are used internally as a mask of access permissions to allow from **ioflag**.

#### path_len
The length of the **path** member.

#### eflag
Extended flags:

- **_IO_CONNECT_EFLAG_DIR** — the path referenced a directory.
- **_IO_CONNECT_EFLAG_DOT** — the last component of a path was . or .. (i.e. the current or parent directory).

#### extra_type
One of:

- **_IO_CONNECT_EXTRA_NONE**
- **_IO_CONNECT_EXTRA_LINK**
- **_IO_CONNECT_EXTRA_SYMLINK**
- **_IO_CONNECT_EXTRA_MQUEUE**
The length of any extra data included in the message.

The path that the client is trying to connect to, relative to the resource manager’s mountpoint.

Classification:

QNX Neutrino

See also:

```
_io_connect_ftype_reply, _io_connect_link_reply,
resmgr_connect_funcs_t
```

“The _IO_OPEN message for filesystems” in the Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```c
struct _io_connect_ftype_reply {
    uint16_t status;
    uint16_t reserved;
    uint32_t file_type;
};
```

Description:

A resource manager uses the `io_connect_ftype_reply` structure to send a status and a file type to a client that has sent a connect message.

The members include:

- **status**: Typically one of the `errno` values.

- **file_type**: The file type; one of the following (defined in `<sys/ftype.h>`):
  - `_FTYPE_ANY` — the path name can be anything.
  - `_FTYPE_LINK` — reserved for the Process Manager.
  - `_FTYPE_MOUNT` — receive mount requests on the path (path must be NULL).
  - `_FTYPE_MQUEUE` — reserved for a message-queue manager.
  - `_FTYPEPIPE` — reserved for a pipe manager.
  - `_FTYPE_SEM` — reserved for a semaphore manager.
  - `_FTYPE_SHMEM` — reserved for a shared memory object.
  - `_FTYPE_SOCKET` — reserved for a socket manager.
  - `_FTYPE_SYMLINK` — reserved for the Process Manager.
Classification:

QNX Neutrino

See also:

_io_connect, _io_connect_link_reply, resmgr_connect_funcs_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```c
struct _io_connect_link_reply {
    uint32_t reserved1;
    uint32_t file_type;
    uint8_t eflag;
    uint8_t reserved2[1];
    uint16_t chroot_len;
    uint32_t umask;
    uint16_t nentries;
    uint16_t path_len;

    /*
       struct _io_connect_entry server[nentries];
       char path[path_len];
    or
       struct _server_info info;
       io_?_t msg;
    */
};
```

Description:

A resource manager uses the `io_connect_link_reply` structure in a reply to a client that redirects the client to another resource. The members include:

- `file_type`: The file type; one of the following (defined in `<sys/ftype.h>`):
  - `_FTYPE_ANY` — the path name can be anything.
  - `_FTYPE_LINK` — reserved for the Process Manager.
  - `_FTYPE_MOUNT` — receive mount requests on the path (`path` must be NULL).
  - `_FTYPE_MQUEUE` — reserved for a message-queue manager.
  - `_FTYPE_PIPE` — reserved for a pipe manager.
  - `_FTYPE_SEM` — reserved for a semaphore manager.
- **_FTYPE_SHMEM** — reserved for a shared memory object.
- **_FTYPE_SOCKET** — reserved for a socket manager.
- **_FTYPE_SYMLINK** — reserved for the Process Manager.

**eflag**

Extended flags:
- **_IO_CONNECT_EFLAG_DIR** — the path referenced a directory.
- **_IO_CONNECT_EFLAG_DOT** — the last component of a path was . or .. (i.e. the current or parent directory).

**chroot_len**

The length of chroot in the returned path.

**umask**

One of:
- **S_IFBLK** — block special.
- **S_IFCHR** — character special.
- **S_IFDIR** — directory.
- **S_IFIFO** — FIFO special.
- **S_IFLNK** — symbolic link.
- **S_IFMT** — type of file.
- **S_IFNAM** — special named file.
- **S_IFREG** — regular.
- **S_IFSOCK** — socket.

**nentries**

If this member is zero, the path is a symbolic link.

**path_len**

The length of the path including the terminating null character. If this member is zero, the path is null-terminated.
Classification:

QNX Neutrino

See also:

_io_connect, _io_connect_ftype_reply,
resmgr_connect_funcs_t

Writing a Resource Manager chapter of the *Programmer's Guide.*
ioctl()© 2007, QNX Software Systems GmbH & Co. KG.
Control a device

Synopsis:

```c
#include <sys/ioctl.h>

int ioctl( int fd,
           int request,
           ... );
```

Arguments:

- `fd`  
  An open file descriptor for the file or device that you want to manipulate.

- `request`  
  What you want to do to the file. The macros and definitions that you use in specifying a request are located in the file `<sys/ioctl.h>`.

Additional arguments

As required by the request.

Library:

```
libc
```

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ioctl()` function manipulates the underlying parameters of files. In particular, it can be used to control many of the operating attributes of files (such as the attributes of terminals).

The `request` argument determines whether the subsequent arguments are an “in” or “out” parameter; it also specifies the size of the arguments in bytes.
Returns:
A value based on the `request`, or -1 if an error occurs (`errno` is set).

Errors:

- EBADF: Invalid descriptor `fd`.
- EINVAL: The `request` or optional variables aren’t valid.
- ENOTTY: The `fd` argument isn’t associated with a character special device; the specified `request` doesn’t apply to the kind of object that the descriptor `fd` references.

Classification:

POSIX 1003.1 XSR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

The `ioctl()` function is a Unix function that varies greatly from platform to platform.

See also:

`devctl()`
Retrieve server attributes

Synopsis:

```c
#include <sys/iomgr.h>

int iofdinfo( int filedes,
              unsigned flags,
              struct _fdinfo * info,
              char * path,
              int maxlen );
```

Arguments:

- `filedes` A file descriptor for the connection that you want to query.
- `flags` Specify _FDINFO_FLAG_LOCALPATH to return only the local path info (i.e. exclude the network path info).
- `info` NULL, or a pointer to an _fdinfo structure that contains the connection information defined in `<sys/iomgr.h>`. Specify NULL if it’s not required.
- `path` A pointer to a buffer where the function can store the path associated with the file descriptor. Specify NULL if it’s not required.
- `maxlen` The length of the buffer pointed to `path`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofdinfo()` function retrieves the server’s attribute information for the connection referred to by `filedes`. 

© 2007, QNX Software Systems GmbH & Co. KG.
Returns:

The length of the associated *filedes* pathname, or -1 if an error occurs (*errno* is set).

Errors:

- **EFAULT** A fault occurred in a server’s address space when it tried to access the caller’s message buffers.
- **EMSGSIZE** Insufficient space available in the server’s buffer for the *fdinfo* data structure.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`iofunc_fdinfo()`, `iofunc_fdinfo_default()`, `resmgr_pathname()`
Initialize the default attribute structure

Synopsis:
```
#include <sys/iofunc.h>

void iofunc_attr_init ( iofunc_attr_t *attr,  
                     mode_t mode,  
                     iofunc_attr_t *dattr,  
                     struct _client_info *info );
```

Arguments:

- `attr` A pointer to the `iofunc_attr_t` structure that you want to initialize.
- `mode` The type and access permissions that you want to use for the resource. For more information, see “Access permissions” in the documentation for `stat()`.
- `dattr` NULL, or a pointer to a `iofunc_attr_t` structure that you want to use to initialize the structure pointed to by `attr`.
- `info` NULL, or a pointer to a `client_info` structure that contains the information about a client connection. For information about this structure, see `ConnectClientInfo()`.

Library:

- `libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_attr_init()` function initializes the passed `attr` structure with the information derived from the optional `dattr`, the `mode`, and the user and group IDs from the optional `info` client information structure.

The `count`, `rcount`, `wcount`, `rlocks` and `wlocks` counters are reset to zero in the `iofunc_attr_t` structure that `attr` points to.
### iofunc_attr_init()

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_attr_lock()`, `iofunc_attr_t`, `iofunc_attr_unlock()`, `iofunc_ocb_attach()`, `iofunc_ocb_detach()`, `resmgr_attach()`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
**iofunc_attr_lock()**

Lock the attribute structure

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_attr_lock( iofunc_attr_t *attr );
```

**Arguments:**

- `attr` A pointer to the `iofunc_attr_t` structure that you want to lock.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_attr_lock()` function locks the attribute structure that `attr` points to, preventing other threads in the resource manager from changing information.

Call this function (or `iofunc_attr_trylock()`) before you make any modifications to the attribute structure. After you're finished making modifications, call `iofunc_attr_unlock()` to release the lock.

Note that this is a *counting* locking mechanism. This means that a given thread can lock the attributes structure multiple times; it must then unlock the attributes structure a corresponding number of times in order to have the attributes structure considered unlocked. If another thread attempts to lock the structure while a thread has the structure locked, the other thread blocks.

**Returns:**

- `EOK` Success.
- `EAGAIN` On the first use, all kernel mutex objects were in use.
iofunc_attr_lock()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

iofunc_attr_init(), iofunc_attr_t, iofunc_attr_trylock(), iofunc_attr_unlock()

Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

#include <sys/iofunc.h>

typedef struct _iofunc_attr {
    IOFUNC_MOUNT_T *mount;
    uint32_t flags;
    int32_t lock_tid;
    uint16_t lock_count;
    uint16_t count;
    uint16_t rcount;
    uint16_t wcount;
    uint16_t rlocks;
    uint16_t wlocks;
    struct _iofunc_mmap_list *mmap_list;
    struct _iofunc_lock_list *lock_list;
    void *list;
    uint32_t list_size;
#if !defined(_IOFUNC_OFFSET_BITS) || _IOFUNC_OFFSET_BITS == 64
        #if _FILE_OFFSET_BITS - 0 == 64
            off_t nbytes;
            ino_t inode;
        #else
            off64_t nbytes;
            ino64_t inode;
        #endif
    #elif _IOFUNC_OFFSET_BITS - 0 == 32
        #if !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
            #if defined(__LITTLEENDIAN__)
                off_t nbytes;
                off_t nbytes_hi;
                ino_t inode;
                ino_t inode_hi;
            #else
                off_t nbytes_hi;
                off_t nbytes;
                ino_t inode_hi;
                ino_t inode;
            #endif
        #elif defined(__BIGENDIAN__)
            off_t nbytes_hi;
            off_t nbytes;
            ino_t inode_hi;
            ino_t inode;
        #else
            #error endian not configured for system
        #endif
    #endif
    #elif _IOFUNC_OFFSET_BITS - 0 == 32
        #if !_FILE_OFFSET_BITS || _FILE_OFFSET_BITS == 32
            #if defined(__LITTLEENDIAN__)
                #endif
            #else
                #endif
        #endif
    #else
        #error endian not configured for system
    #endif
#if defined(__LITTLEENDIAN__)
    #endif
#else
    #error endian not configured for system
#endif
The `iofunc_attr_t` structure describes the attributes of the device that’s associated with a resource manager. The members include the following:

- **mount**: A pointer to a structure containing information about the mountpoint. By default, this structure is of type `iofunc_mount_t`, but you can specify your own structure by changing the IOFUNC_MOUNT_T manifest.

- **flags**: Flags that your resource manager can set to indicate the state of the device. This member is a combination of the following flags:
IOFUNC_ATTR_ATIME
    The access time is no longer valid. Typically set on a read from the resource.

IOFUNC_ATTR_CTIME
    The change of status time is no longer valid. Typically set on a file info change.

IOFUNC_ATTR_DIRTY_NLINK
    The number of links has changed.

IOFUNC_ATTR_DIRTY_MODE
    The mode has changed.

IOFUNC_ATTR_DIRTY_OWNER
    The uid or the gid has changed.

IOFUNC_ATTR_DIRTY_RDEV
    The rdev member has changed, e.g. mknod().

IOFUNC_ATTR_DIRTY_SIZE
    The size has changed.

IOFUNC_ATTR_DIRTY_TIME
    One or more of mtime, atime, or ctime has changed.

IOFUNC_ATTR_MTIME
    The modification time is no longer valid. Typically set on a write to the resource.

In addition to the above, your resource manager can use in any way the bits in the range defined by IOFUNC_ATTR_PRIVATE (see <sys/iofunc.h>).

lock_tid
    The ID of the thread that has locked the attributes. To support multiple threads in your resource manager, you’ll need to lock the attribute structure so that only one thread at a time is allowed to change it.

The resource manager layer automatically locks the attribute (using iofunc_attr_lock()) for you when certain handler functions are called (i.e. IO_*).


text
A thread must unlock the attributes as many times as it locked them.

lock_count The number of times the thread has locked the attribute structure. You can lock the attributes by calling `iofunc_attr_lock()` or `iofunc_attr_trylock()`; unlock them by calling `iofunc_attr_unlock()`.

count The number of OCBs using this attribute in any manner. When this count is zero, no one is using this attribute.

rcount The number of OCBs using this attribute for reading.

wcount The number of OCBs using this attribute for writing.

rlocks The number of read locks currently registered on the attribute.

wlocks The number of write locks currently registered on the attribute.

mmap_list and lock_list

To manage their particular functionality on the resource, the `mmap_list` member is used by `iofunc_mmap()` and `iofunc_mmap_default()`; the `lock_list` member is used by `iofunc_lock_default()`. Generally, you shouldn’t need to modify or examine these members.

list Reserved for future use.

list_size Size of reserved area; reserved for future use.

nbytes The number of bytes in the resource; your resource manager can change this value.

For a file, this would contain the file’s size. For special devices (e.g. `/dev/null`) that don’t support `lseek()` or have a radically different interpretation for `lseek()`, this field isn’t used (because you wouldn’t use
any of the helper functions, but would supply your own instead.) In these cases, we recommend that you set this field to zero, unless there’s a meaningful interpretation that you care to put to it.

*inode*

This is a mountpoint-specific inode that must be unique per mountpoint. You can specify your own value, or 0 to have the Process manager fill it in for you. For filesystem type of applications, this may correspond to some on-disk structure. In any case, the interpretation of this field is up to you.

*uid* and *gid*

The user ID and group ID of the owner of this resource. These fields are updated automatically by the `chown()` helper functions (e.g. `iofunc_chown_default()`) and are referenced in conjunction with the *mode* member for access-granting purposes by the `open()` help functions (e.g. `iofunc_open_default()`).

*mtime*, *atime*, and *ctime*

POSIX time members:

- *mtime* — modification time (`write()` updates this).
- *atime* — access time (`read()` updates this).
- *ctime* — change of status time (`write()`, `chmod()` and `chown()` update this).

One or more of the three time members may be invalidated as a result of calling an iofunc-layer function. To see if a time member is invalid, check the *flags* member. This is to avoid having each and every I/O message handler go to the kernel and request the current time of day, just to fill in the attribute structure’s time member(s).

To fill the members with the correct time, call `iofunc_time_update()`.

*mode*

The resource’s mode (e.g. type, permissions). Valid modes may be selected from the S_* series of
constants in `<sys/stat.h>`; see “Access permissions” in the documentation for `stat()`.

nlink  
The number of links to this particular name; your resource manager can modify this member. For names that represent a directory, this value must be greater than 2.

rdev  
The device number for a character special device and the `rdev` number for a named special device.

**Classification:**

QNX Neutrino

**See also:**

`iofunc_attr_lock()`, `iofunc_attr_trylock()`, `iofunc_attr_unlock()`, `iofunc_lock_default()`, `iofunc_mmap()`, `iofunc_mmap_default()`, `iofunc_ocb_t`, `iofunc_time_update()`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
**iofunc_attr_trylock()**

Try to lock the attribute structure

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_attr_trylock( iofunc_attr_t *attr );
```

**Arguments:**

- `attr` A pointer to the `iofunc_attr_t` structure that you want to lock.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_attr_trylock()` function attempts to lock the attribute structure `attr`, preventing other threads in the resource manager from changing information. If it can’t lock `attr` immediately, it returns EBUSY.

Call this function (or `iofunc_attr_lock()`) before you make any modifications to the attribute structure. After you’re finished making modifications, call `iofunc_attr_unlock()` to release the lock.

Note that this is a counting locking mechanism. This means that a given thread can lock the attributes structure multiple times; it must then unlock the attributes structure a corresponding number of times in order to have the attributes structure considered unlocked. If another thread attempts to lock the structure while a thread has the structure locked, the other thread will block.

**Returns:**

- EOK Success.
- EBUSY The calling thread couldn’t lock the attributes immediately.
iofunc_attr_trylock()

EAGAIN   On the first use, all kernel mutex objects were in use.

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

iofunc_attr_init(), iofunc_attr_lock(), iofunc_attr_t, iofunc_attr_unlock()

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Unlock the attribute structure

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_attr_unlock( iofunc_attr_t *attr );
```

Arguments:

- `attr` A pointer to the `iofunc_attr_t` structure that you want to unlock.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_attr_unlock()` function unlocks the attribute structure `attr`, allowing other threads in the resource manager to change information.

Use this function in conjunction with `iofunc_attr_lock()` or `iofunc_attr_trylock()`. Call `iofunc_attr_unlock()` after you’ve made modifications to the attribute structure. You must unlock the structure as many times as you locked it.

Returns:

- `EOK` Success.
- `EAGAIN` On the first use, all kernel mutex objects were in use.

Classification:

QNX Neutrino
iofunc_attr_unlock()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- iofunc_attr_init(), iofunc_attr_lock(), iofunc_attr_t,
- iofunc_attr_trylock()

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**iofunc_check_access()**  © 2007, QNX Software Systems GmbH & Co. KG.

Check access permissions

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_check_access(
    resmgr_context_t *ctp,
    const iofunc_attr_t *attr,
    mode_t checkmode,
    const struct _client_info *info);
```

**Arguments:**

- `ctp`  
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- `attr`  
  A pointer to the `iofunc_attr_t` structure that defines the characteristics of the device that’s associated with the resource manager.

- `checkmode`  
  The type and access permissions that you want to check for the resource. For more information, see below.

- `info`  
  A pointer to a `_client_info` structure that contains the information about a client connection. For information about this structure, see `ConnectClientInfo()`. You can get this structure by calling `iofunc_client_info()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `iofunc_check_access()` function verifies that the client is allowed access to the resource, as specified by a combination of who the client is (`info`), and the resource attributes `attr->mode`, `attr->uid` and `attr->gid`. Access is tested based upon the `checkmode` parameter.

The `checkmode` parameter determines which checks are done. It’s a bitwise OR of the following constants:

- **S_ISUID**: Verifies that the effective user ID of the client is equal to the user ID specified by the `attr->uid` member.
- **S_ISGID**: Verifies that the effective group ID or one of the supplementary group IDs of the client is equal to the group ID specified by the `attr->gid` member.
- **S_IREAD**: Verifies that the client has READ access to the resource as specified by `attr->mode`.
  
  If the client’s effective user ID matches that of `attr->uid`, then the permission check is made against the owner permission field of `attr->mode` (mask 0700 octal).
  
  If the client’s effective user ID doesn’t match that of `attr->uid`, then if the client’s effective group ID matches that of `attr->gid`, or one of the client’s supplementary group IDs matches `attr->gid`, the check is made against the group permission field of `attr->mode` (mask 0070 octal).
  
  If none of the group fields match, the check is made against the other permission field of `attr->mode` (mask 0007 octal).

- **S_IWRITE**: Same as `S_IREAD`, except WRITE access is tested.

- **S_IEXEC**: Same as `S_IREAD`, except EXECUTE access is tested. Note that since most resource managers don’t actually execute code, the execute access is typically used in its
directory sense, i.e. to test for directory accessibility, rather than execute access.

The S_ISUID and S_ISGID flags are mutually exclusive, that is, you may specify at most one of them. In conjunction with the S_ISUID and S_ISGID flags, you may specify zero or more of the S_IREAD, S_IWRITE, and S_IEXEC flags. If no flags are specified, the permission checks are performed for privileged (root) access.

Here's some pseudo-code to try to explain this:

```c
if superuser:
    return EOK

if S_ISUID and effective user ID == file user ID:
    return EOK

if S_ISGID and effective group ID == file group ID:
    return EOK

if S_IREAD or S_IWRITE or S_IEXEC:
    if caller’s user ID == effective user ID:
        if all permissions are set in file’s owner mode bits:
            return EOK
        else:
            return EACCESS
    else:
        return EACCESS

if ( caller’s group ID or supplementary group IDs ) ==
effective group ID:
    if all permissions are set in file’s group mode bits:
        return EOK
    else:
        return EACCESS

    if all permissions are set in file’s other mode bits:
        return EOK
    else:
        return EACCESS

return EPERM
```
Returns:

- **EACCES**: The client doesn’t have permissions to do the operation.
- **ENOSYS**: NULL was passed for `info` structure.
- **EOK**: Successful completion.
- **EPERM**: The group ID or owner ID didn’t match.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_client_info()`, `iofunc_open()`, `iofunc_read_verify()`, `iofunc_write_verify()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

September 10, 2007
**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_chmod ( resmgr_context_t *ctp,
                   io_chmod_t *msg,
                   iofunc_ocb_t *ocb,
                   iofunc_attr_t *attr );
```

**Arguments:**

- `ctp`: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg`: A pointer to the `io_chmod_t` structure that contains the message that the resource manager received; see below.
- `ocb`: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `attr`: A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_chmod()` helper function implements POSIX semantics for the client’s `chmod()` call, which is received as an _IO_CHMOD message by the resource manager.

The `iofunc_chmod()` function verifies that the client has the necessary permissions to effect a `chmod()` on the attribute. If so, the `chmod()` is
performed, modifying elements of the `ocb->attr` structure. This function takes care of updating the `IOFUNC_ATTR_CTIME`, `IOFUNC_ATTR_DIRTY_TIME`, and `IOFUNC_ATTR_DIRTY_MODE` bits in `ocb->attr->flags`. You can use `iofunc_time_update()`, to update the appropriate time fields in `ocb->attr`.

You can use `iofunc_chmod()`, for example, in a filesystem manager, where an `_IO_CHMOD` message was received, and the filesystem code must now write the values to the medium. The filesystem code may wish to block the client thread until the data was actually written to the medium. Contrast this scenario to the behavior of `iofunc_chmod_default()`, which calls this routine, and replies to the client thread.

**io_chmod_t structure**

The `io_chmod_t` structure holds the `_IO_CHMOD` message received by the resource manager:

```c
struct _io_chmod {
    uint16_t type;
    uint16_t combine_len;
    mode_t mode;
};
```

```c
typedef union {
    struct _io_chmod i;
} io_chmod_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, `i`.

The `i` member is a structure of type `_io_chmod` that contains the following members:

- **type** `_IO_CHMOD`
- **combine_len** If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the *Programmer’s Guide*. 
### iofunc_chmod()

*mode* The new mode. For more information, see “Access permissions” in the documentation for *stat()*.  

**Returns:**

- **EOK** Successful completion.
- **EROFs** An attempt was made to chmod on a read-only filesystem.
- **EACCES** The client doesn’t have permissions to do the operation.
- **EPERM** The group ID or owner ID didn’t match.

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

- iofunc_attr_t, iofunc_chmod_default(), iofunc_ocb_t, iofunc_time_update(), resmgr_context_t

Writing a Resource Manager chapter of the* Programmer’s Guide.*
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_chmod_default( resmgr_context_t *ctp,
    io_chmod_t *msg,
    iofunc_ocb_t *ocb);
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg**: A pointer to the `io_chmod_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_chmod()`.
- **ocb**: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_chmod_default()` function implements POSIX semantics for the client’s `chmod()` call, which is received as an `_IO_CHMOD` message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `chmod` position, or you can call `iofunc_func_init()` to initialize all the functions to their default values.

The `iofunc_chmod_default()` function calls `iofunc_chmod()` to do the actual work.
Returns:

EOK       Successful completion.
EROFS     An attempt was made to chmod() on a read-only filesystem.
EACCES    The client doesn’t have permissions to do the operation.
EPERM     The group ID or owner ID didn’t match.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

iofunc_chmod(), iofunc_func_init(), iofunc_ocb_t, iofunc_time_update(), resmgr_attach(), resmgr_context_t, resmgr_io_funcs_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```
#include <sys/iofunc.h>

int iofunc_chown ( resmgr_context_t *ctp,
                   io_chown_t *msg,
                   iofunc_ocb_t *ocb,
                   iofunc_attr_t *attr );
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg**: A pointer to the `io_chown_t` structure that contains the message that the resource manager received; see below.
- **ocb**: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- **attr**: A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_chown()` helper function implements POSIX semantics for the client’s `chown()` call, which is received as an _IO_CHOWN message by the resource manager. The `iofunc_chown()` function verifies that the client has the necessary permissions to effect a chown on the attribute. If so, the chown is
iofunc_chown() performed, modifying elements of the ocb->attr structure. As per POSIX 1003.1, if the client isn’t root, iofunc_chown() clears the set-user-id and set-group-id bits in the ocb->attr->mode member.

This function takes care of updating the IOFUNC_ATTR_CTIME, IOFUNC_ATTR_DIRTY_TIME, and IOFUNC_ATTR_DIRTY_MODE bits in ocb->attr->flags. You can use iofunc_time_update(), to update the appropriate time fields in ocb->attr.

io_chown_t structure

The io_chown_t structure holds the _IO_CHOWN message received by the resource manager:

```c
struct _io_chown {
    uint16_t type;
    uint16_t combine_len;
    int32_t gid;
    int32_t uid;
};
```

typedef union {
    struct _io_chown i;
} io_chown_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, i.

The i member is a structure of type _io_chown that contains the following members:

- **type** _IO_CHOWN.
- **combine_len** If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **gid** The new group ID.
- **uid** The new user ID.
Returns:

- **EOK**  Successful completion.
- **EROFS**  An attempt was made to chown on a read-only filesystem.
- **EACCES**  The client doesn’t have permissions to do the operation.
- **EPERM**  The group ID or owner ID didn’t match.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_attr_t`, `iofunc_chmod()`, `iofunc_chown_default()`,
- `iofunc_ocb_t`, `iofunc_time_update()`, `resmgr_attach()`,
- `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

September 10, 2007
#include <sys/iofunc.h>

int iofunc_chown_default( resmgr_context_t *ctp,  
                        io_chown_t *msg,  
                        iofunc_ocb_t *ocb );

Arguments:

ctp  A pointer to a resmgr_context_t structure that the  
     resource-manager library uses to pass context information  
     between functions.

msg A pointer to the io_chown_t structure that contains the  
     message that the resource manager received. For more  
     information, see the documentation for iofunc_chown().

ocb A pointer to the iofunc_ocb_t structure for the Open  
     Control Block that was created when the client opened the  
     resource.

Library:

libc

Use the -l c option to qcc to link against this library. This library is  
usually included automatically.

Description:

The iofunc_chown_default() function implements POSIX semantics  
for the client’s chown() call, which is received as an _IO_CHOWN  
message by the resource manager.

You can place this function directly into the io_funcs table passed to  
resmgr_attach(), at the chown position, or you can call  
iofunc_func_init() to initialize all of the functions to their default  
values.

The iofunc_chown_default() function calls iofunc_chown() to do the  
actual work.
Returns:

- **EOK**: Successful completion.
- **EROFS**: An attempt was made to chown on a read-only filesystem.
- **EACCES**: The client doesn’t have permissions to do the operation.
- **EPERM**: The group ID or owner ID didn’t match.

Classification:

**QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_chown()`, `iofunc_func_init()`, `iofunc_ocb_t`,
- `iofunc_time_update()`, `resmgr_attach()`, `resmgr_context_t`,
- `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

---

© 2007, QNX Software Systems GmbH & Co. KG.
iofunc_client_info() — Return information about a client connection

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_client_info ( resmgr_context_t * ctp,
                        int ioflag,
                        struct _client_info * info );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- `ioflag` Zero, or the constant O_REALIDS. This argument is passed in the _IO_OPEN message during an open request. If O_REALIDS is specified, `iofunc_client_info()` swaps the real and effective values of the user and group IDs before returning. This is a QNX Neutrino extension, to swap real and effective user and group IDs in an atomic operation.

- `info` A pointer to a `_client_info` structure that the function fills with information about a client connection. For information about this structure, see `ConnectClientInfo()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_client_info()` function fetches the `info` structure for the client. It calls `ConnectClientInfo()` to gather the information, based on the server connection ID found in `ctp->info.scoid`.
Returns:

- EFAULT: A fault occurred when the kernel tried to access the info buffer provided.
- EINVAL: The client process is no longer valid.
- EOK: Successful completion.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
</tr>
<tr>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ConnectClientInfo()

Writing a Resource Manager chapter of the Programmer’s Guide.
Free all locks allocated for the client process

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_close_dup( resmgr_context_t* ctp,
                      io_close_t* msg,
                      iofunc_ocb_t* ocb,
                      iofunc_attr_t* attr );
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg**: A pointer to the `io_close_t` structure that contains the message that the resource manager received; see below.
- **ocb**: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- **attr**: A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_close_dup()` helper function handles a `_IO_CLOSE` message. This function frees all locks allocated for the client process on the file descriptor and performs any POSIX-related cleanup required when a duplicated `ocb` is detached.
**iofunc_close_dup()**

### io_close_t structure

The `io_close_t` structure holds the _IO_CLOSE message received by the resource manager:

```c
struct _io_close {
    uint16_t type;
    uint16_t combine_len;
};
```

```c
typedef union {
    struct _io_close i;
} io_close_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, `i`.

The `i` member is a structure of type `_io_close` that contains the following members:

- **type**
  - `_IO_CLOSE`.

- **combine_len**
  - If the message is a combine message, `_IO_COMBINE_FLAG` is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the *Programmer’s Guide*.

### Returns:

- **EOK** Success.
- Anything else An error occurred.

### Classification:

QNX Neutrino
See also:

`iofunc_close_dup_default()`, `iofunc_close_ocb()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_close_dup_default(
    resmgr_context_t *ctp,
    io_close_t *msg,
    iofunc_ocb_t *ocb);
```

Arguments:

ctp  A pointer to a `resmgr_context_t` structure that the
     resource-manager library uses to pass context information
     between functions.

msg  A pointer to the `io_close_t` structure that contains the
     message that the resource manager received. For more
     information, see the documentation for `iofunc_close_dup()`.

ocb  A pointer to the `iofunc_ocb_t` structure for the Open
     Control Block that was created when the client opened the
     resource.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is
usually included automatically.

Description:

The `iofunc_close_dup_default()` function implements default actions
for the `IO_CLOSE` message. This function simply calls
`iofunc_close_dup()` to do the actual work.

You can place `iofunc_close_dup_default()` directly into the `io_funcs`
table passed to `resmgr_attach()`, at the `close_dup` position, or you can
call `iofunc_func_init()` to initialize all of the functions to their default
values.
If your resource manager uses `iofunc_lock_default()`, you must use both this function (`iofunc_close_dup_default()`) and `iofunc_unblock_default()`, as they provide necessary ancillary functionality for managing file locks. This is because file locks are owned by the process, and aren’t inherited by the child process.

**Returns:**

EOK  Successful completion.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`iofunc_close_dup()`, `iofunc_func_init()`, `iofunc_ocb_t`, `iofunc_time_update()`, `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_close_ocb( resmgr_context_t* ctp, 
                      iofunc_ocb_t* ocb, 
                      iofunc_attr_t* attr );
```

**Arguments:**

- **ctp**  
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **ocb**  
  A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

- **attr**  
  A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

**Library:**

- **libc**  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_close_ocb()` function detaches the OCB specified by `ocb`, and releases the memory associated with it.
This function assumes that *ocb* points to a `iofunc_ocb_t`. If you encapsulate `iofunc_ocb_t` in your own OCB it must be the first field of your OCB; otherwise, you can’t call this function. If you provide an `ocb_free()` function in the mount structure then it’s called at this point. This means that at least the `iofunc_ocb_t` portion of your OCB is no longer valid after `iofunc_close_ocb()` returns.

The `iofunc_close_ocb()` function calls `iofunc_ocb_detach()` on your behalf.

**Returns:**

- EOK Success.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_attr_t`, `iofunc_close_dup()`, `iofunc_close_ocb_default()`,
- `iofunc_ocb_free()`, `iofunc_ocb_t`, `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_close_ocb_default( resmgr_context_t* ctp,
        void* reserved,
        iofunc_ocb_t* ocb );
```

Arguments:

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **reserved** This argument must be passed as NULL.
- **ocb** A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_close_ocb_default()` function detaches the OCB specified by `ocb`, and releases the memory associated with it.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `close_ocb` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.
This function assumes that `ocb` points to an `iofunc_ocb_t`. If you encapsulate `iofunc_ocb_t` in your own OCB, it must be the first field of your OCB; otherwise, you can’t call this function. If you provide an `ocb_free()` function in the mount structure, it’s called at this point. This means that at least the `iofunc_ocb_t` portion of your OCB is no longer valid after `iofunc_close_ocb()` returns.

The `iofunc_close_ocb_default()` function calls `iofunc_close_ocb()` to do the actual work.

**Returns:**

EOK Success.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`iofunc_close_ocb()`, `iofunc_func_init()`, `iofunc_ocb_t`, `iofunc_time_update()`, `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_devctl( resmgr_context_t *ctp,
                   io_devctl_t *msg,
                   iofunc_ocb_t *ocb,
                   iofunc_attr_t *attr );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_devctl_t` structure that contains the message that the resource manager received; see below.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `attr` A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_devctl()` helper function implements POSIX semantics for the client’s `devctl()` call, which is received as an _IO_DEVCTL message by the resource manager. This function handles the DCMD_ALL* functionality.

This function handles at least the following device control messages:
DCMD_ALL_GETFLAGS

Implements the functionality of the `fcntl()` get-flags command.

DCMD_ALL_SETFLAGS

Implements the functionality of the `fcntl()` set-flags command.

DCMD_ALL_GETMOUNTFLAGS

Returns the mount flag (`mount->flags`) for a resource that has a mount structure defined, else returns a mount flag of zero.

The supported mount flags (bitmask values) for DCMD_ALL_GETMOUNTFLAGS include:

- `_MOUNT_READONLY`
  Read only.

- `_MOUNT_NOEXEC`
  Can’t exec from filesystem.

- `_MOUNT_NOSUID`
  Don’t honor setuid bits on filesystem.

Any other device control messages return ENOTTY.

**io_devctl_t structure**

The `io_devctl_t` structure holds the `_IO_` message received by the resource manager:

```c
struct __io_devctl {
    uint16_t type;
    uint16_t combine_len;
    int32_t dcmd;
    int32_t nbytes;
    int32_t zero;
    /* char data[nbytes]; */
};

struct __io_devctl_reply {
    uint32_t zero;
};
```
typedef union {
    struct _io_devctl i;
    struct _io_devctl_reply o;
} io_devctl_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The i member is a structure of type _io_devctl that contains the following members:

- **type** _IO_DEVCTL_
- **combine_len** If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **dcmd** The device-control command to execute.
- **nbytes** The number of bytes of data being passed with the command.

The commented-out declaration for data indicates that nbytes bytes of data immediately follow the _io_devctl structure.

The _DEVCTL_DATA() macro gets a pointer to the data that follows the message. Call it like this:

```c
data = _DEVCTL_DATA(msg->i);
```

The o member of the io_devctl_t message is a structure of type _io_devctl_reply that contains the following members:

- **ret_val** The value returned by the command.
The number of bytes of data being returned. The commented-out declaration for `data` indicates that `nbytes` bytes of data immediately follow the `_io_devctl_reply` structure.

**Returns:**

- **EOK** Successful completion.
- **EINVAL** An attempt to set the flags for a resource that is synchronized, with no mount structure defined, or no synchronized I/O defined.
- **ENOTTY** An unsupported device control message was decoded.

**Classification:**

- **QNX Neutrino**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `fcntl()`, `iofunc_attr_t`, `iofunc_devctl_default()`, `iofunc_ocb_t`, `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
### io\texttt{func} devctl default()

**Default handler for _IO_DEVCTL messages**

#### Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_devctl_default( resmgr_context_t *ctp,
                           io_devctl_t *msg,
                           iofunc_ocb_t *ocb );
```

#### Arguments:

- **ctp**
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg**
  A pointer to the `io_devctl_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_devctl()`.

- **ocb**
  A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

#### Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

#### Description:

The `iofunc_devctl_default()` function implements the default actions for _IO_DEVCTL messages.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the devctl position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_devctl_default()` function calls `iofunc_devctl()` to do the actual work.
iofunc_devctl_default()

Returns:

- **EOK**  
  Successful completion.

- **EINVAL**  
  An attempt to set the flags for a resource that is synchronized, with no mount structure defined, or no synchronized I/O defined.

- **ENOTTY**  
  An unsupported device control message was decoded.

- **RESMGR_DEFAULT**  
  An supported device control message that isn’t a known DCMD_ALL_* command was decoded.

Classification:

QNX Neutrino

### Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_devctl()`, `iofunc_func_init()`, `iofunc_ocb_t`,
- `iofunc_time_update()`, `resmgr_attach()`, `resmgr_context_t`,
- `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_fdinfo( resmgr_context_t * ctp,
                   iofunc_ocb_t * ocb,
                   iofunc_attr_t * attr,
                   struct _fdinfo * info );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

- `attr` NULL, or a pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

- `info` A pointer to a `_fdinfo` structure that the function fills with the information. This structure is defined in `<sys/iomgr.h>` as:

```c
struct _fdinfo {
    uint32_t    mode;    /* File mode */
    uint32_t    ioflag;  /* Current io flags */
    uint64_t    offset;  /* Current seek position */
    uint64_t    size;    /* Current size of file */
    uint32_t    flags;   /* _FDINFO_* */
    uint16_t    sflag;   /* Share flags */
    uint16_t    count;   /* File use count */
    uint16_t    rcount;  /* File reader count */
    uint16_t    wcount;  /* File writer count */
    uint16_t    rlocks; /* Number of read locks */
    uint16_t    wlocks; /* Number of write locks */
    uint32_t    zero[6];
};
```

The `_fdinfo` structure is included in the reply part of a `io_fdinfo_t` structure; for more information, see the documentation for `iofunc_fdinfo_default()`.
iofunc_fdinfo()

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_fdinfo()` helper function provides the implementation for the client’s `iofdinfo()` call, which is received as an `IO_FDINFO` message by the resource manager.

The `iofunc_fdinfo()` function transfers the appropriate fields from the `ocb` and `attr` structures to the `info` structure. If `attr` is NULL, then the `attr` information comes from the structure pointed to by `ocb->attr`.

Returns:

- **EOK** Successful completion.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `iofdinfo()`, `iofunc_attr_t`, `iofunc_fdinfo_default()`, `iofunc_ocb_t`, `resmgr_context_t`, `resmgr_pathname()`
- Writing a Resource Manager chapter of the *Programmer’s Guide*.
Synopsis:

```c
#include <sys/iomgr.h>

int iofunc_fdinfo_default( resmgr_context_t * ctp,
                          io_fdinfo_t * msg,
                          iofunc_ocb_t * ocb );
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg**: A pointer to the `io_fdinfo_t` structure that contains the message that the resource manager received; see below.
- **ocb**: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_fdinfo_default()` function provides the default handler for the client’s `iofdinfo()` call, which is received as an _IO_FDINFO message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `fdinfo` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_fdinfo_default()` function calls `iofunc_fdinfo()` and `resmgr_pathname()` to do the actual work.
io_fdinfo_t structure

The io_fdinfo_t structure holds the _IO_FDNINFO message received by the resource manager:

```c
struct _io_fdinfo {
    uint16_t type;
    uint16_t combine_len;
    uint32_t flags;
    int32_t path_len;
    uint32_t reserved;
};

struct _io_fdinfo_reply {
    uint32_t zero[2];
    struct _fdinfo info;
    /* char path[path_len + 1]; */
};

typedef union {
    struct _io_fdinfo i;
    struct _io_fdinfo_reply o;
} io_fdinfo_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The i member is a structure of type _io_fdinfo that contains the following members:

- **type**
  - _IO_FDNINFO.

- **combine_len**
  - If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.

- **flags**
  - Specify _FDINFO_FLAG_LOCALPATH to return only the local path info (i.e. exclude the network path info).

- **path_len**
  - The size of the path reply buffers that follow the reply.
The `o` member is a structure of type `_io_fdinfo_reply` that contains the following members:

```
info A _fdinfo structure that’s defined (in `<sys/iomgr.h>`) as:

struct _fdinfo {
    uint32_t mode; /* File mode */
    uint32_t ioflag; /* Current io flags */
    uint64_t offset; /* Current seek position */
    uint64_t size; /* Current size of file */
    uint32_t flags; /* _FDINFO_* */
    uint16_t sflag; /* Share flags */
    uint16_t count; /* File use count */
    uint16_t rcount; /* File reader count */
    uint16_t wcount; /* File writer count */
    uint16_t rlocks; /* Number of read locks */
    uint16_t wlocks; /* Number of write locks */
    uint32_t zero[6];
};
```

The commented-out declaration for `path` indicates that `path_len + 1` bytes of data immediately follow the `_io_fdinfo_reply` structure.

**Returns:**

The length of the path, or -1 if an error occurs (`errno` is set).

**Errors:**

EMSGSIZE Insufficient space available in the server’s buffer to receive the entire message.

**Classification:**

QNX Neutrino
Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

See also:

- iofdinfo(), iofunc_fdinfo_default(), iofunc_func_init(),
- iofunc_ocb_t, resmgr_attach(), resmgr_context_t,
- resmgr_io_funcs_t, RESMGR_NPARTS(), resmgr_pathname()

Writing a Resource Manager chapter of the Programmer's Guide.
iofunc_func_init()

Initialize the default POSIX-layer function tables

Synopsis:

```c
#include <sys/iofunc.h>

void iofunc_func_init(
    unsigned nconnect,  
    resmgr_connect_funcs_t *connect, 
    unsigned nio,       
    resmgr_io_funcs_t *io );
```

Arguments:

- `nconnect` The number of entries in the `connect` table that you want to fill. Typically, you pass
  `_RESMGR_CONNECT_NFUNCS` for this argument.
- `connect` A pointer to a `resmgr_connect_funcs_t` structure that you want to fill with the default connect functions.
- `nio` The number of entries in the `io` table that you want to fill. Typically, you pass `_RESMGR_IO_NFUNCS` for this argument.
- `io` A pointer to a `resmgr_io_funcs_t` structure that you want to fill with the default I/O functions.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_func_init()` function initializes the passed `connect` and `io` structures with the POSIX-layer default functions. For information about the default functions, see `resmgr_connect_funcs_t` and `resmgr_io_funcs_t`.

The `nconnect` and `nio` arguments indicate how many entries this function should fill. This is in place to support forward compatibility.
Examples:

Fill a connect and I/O function table with the POSIX-layer defaults:

```c
#include <sys/iofunc.h>

static resmgr_connect_funcs_t my_connect_functions;
static resmgr_io_funcs_t my_io_functions;

int main (int argc, char **argv)
{
    ...
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &my_connect_functions,
                      _RESMGR_IO_NFUNCS, &my_io_functions);

    /*
     * At this point, the defaults have been filled in.
     * You may now override some of the default functions with
     * functions that you have written:
     */

    my_io_functions.io_read = my_io_read;

    ...}
```

The above example initializes your connect and I/O function structures (`my_connect_functions` and `my_io_functions`) with the POSIX-layer defaults. If you didn’t override any of the functions, your resource manager would behave like `/dev/null` — any data written to it would be discarded, and an attempt to read data from it would immediately return an EOF.

Since this isn’t desirable in most cases, you’ll often provide functionality for some functions, such as reading, writing, and device control to your device. In the example above, we’ve explicitly supplied our own handler for reading from the device, via a function called `my_io_read()`.

Classification:

QNX Neutrino
### iofunc_func_init()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_attr_init()`, `resmgr_attach()`, `resmgr_connect_funcs_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide.*


**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_link( resmgr_context_t* ctp,
                io_link_t* msg,
                iofunc_attr_t* attr,
                iofunc_attr_t* dattr,
                struct_client_info* info);
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg** A pointer to the `io_link_t` structure that contains the message that the resource manager received; see below.
- **attr** A pointer to the `iofunc_attr_t` structure that describes the characteristics of the resource.
- **dattr** NULL, or a pointer to the `iofunc_attr_t` structure that describes the characteristics of the parent directory.
- **info** NULL, or a pointer to a `struct_client_info` structure that contains information about the client. For information about this structure, see `ConnectClientInfo()`.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_link()` helper function links directory `attr` to `dattr` for context `ctp`. It’s similar to the `iofunc_open()` function:
The `iofunc_link()` function checks to see if the client (described by the optional `info` structure) has access to open the resource (name passed in the `msg` structure). The `attr` structure describes the resource’s attributes, and the optional `dattr` structure defines the attributes of the parent directory (i.e. if `dattr` isn’t NULL, it implies that the resource identified by `attr` is being created within the directory specified by `dattr`).

You can pass the `info` argument as NULL, in which case `iofunc_link()` obtains the client information itself via a call to `iofunc_client_info()`. It is, of course, more efficient to get the client info once, rather than calling this function with NULL every time.

If you pass NULL in `info`, the function returns information about a client’s connection in `info`, and an error constant.

### io_link_t structure

The `io_link_t` structure holds the _IO_CONNECT message received by the resource manager:

```c
typedef union {
    struct _io_connect connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_link_t;
```

This message structure is a union of an input message (coming to the resource manager), _io_connect, and two possible output or reply messages (going back to the client):

- `_io_connect_link_reply` if the reply is redirecting the client to another resource
  
  Or:

- `_io_connect_ftype_reply` if the reply consists of a status and a file type.

The reply includes the following additional information:

```c
struct _io_resmgr_link_extra {
    uint32_t nd;
    int32_t pid;
}
```
int32_t chid;
uint32_t handle;
uint32_t flags;
uint32_t file_type;
uint32_t reserved[2];

typedef union _io_link_extra {
    struct _msg_info info; /* EXTRA_LINK (from client) */
    void *ocb; /* EXTRA_LINK (from resmgr functions) */
    char path[1]; /* EXTRA_SYMLINK */
    struct _io_resmgr_link_extra resmgr; /* EXTRA_RESMGR_LINK */
} io_link_extra_t;

info A pointer to a _msg_info structure.

Returns:

EOK Success.

EBADFSYS. NULL was passed in attr and dattr.

EFAULT A fault occurred when the kernel tried to access the info buffer.

EINVAL The client process is no longer valid.

ENOSYS NULL was passed in info.

EPERM The group ID or owner ID didn’t match.

Classification:

QNX Neutrino

Safety

Cancellation point No
Interrupt handler No

continued...
iofunc_link()

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ConnectClientInfo(), iofunc_attr_t, iofunc_client_info(),
iofunc_open(), _msg_info, resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
**iofunc_lock()**

Lock a resource

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_lock( resmgr_context_t * ctp,
                 io_lock_t * msg,
                 iofunc_ocb_t * ocb,
                 iofunc_attr_t * attr );
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg** A pointer to the `io_lock_t` structure that contains the message that the resource manager received.
- **ocb** A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- **attr** A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that's associated with your resource manager.

**Library:**

-libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The function `iofunc_lock()` does what is required for POSIX locks. This function isn’t currently implemented.
**Returns:**

ENOSYS  

The `iofunc_lock()` function isn’t currently supported.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`iofunc_lockcalloc()`, `iofunc_lockfree()`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
iofunc_lock_calloc() © 2007, QNX Software Systems GmbH & Co. KG.

Allocate memory to lock structures

Synopsis:

```c
#include <sys/iofunc.h>

iofunc_lock_list_t *iofunc_lock_calloc
    ( resmgr_context_t *ctp,
      IOFUNC_OCB_T *ocb,
      size_t size );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `ocb` A pointer to the the Open Control Block (typically a `iofunc_ocb_t` structure) that was created when the client opened the resource.
- `size` The amount of memory that you want to allocate.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `iofunc_lock_calloc()` is used by `iofunc_lock()` to allocate memory to lock structures.

Returns:

A pointer to a zeroed buffer that the POSIX layer uses for locks, or `NULL` if no memory could be allocated.
iofunc_lock_calloc()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_lock(), iofunc_lock_free()

Writing a Resource Manager chapter of the Programmer’s Guide.
**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_lock_default( resmgr_context_t *ctp,
     io_lock_t *msg,
     iofunc_ocb_t *ocb );
```

**Arguments:**

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_lock_t` structure that contains the message that the resource manager received; see `iofunc_lock()`.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_lock_default()` function implements POSIX semantics for the _IO_LOCK message (generated as a result of a client `fcntl()` call).

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `lock` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_lock_default()` function verifies that the client has the necessary permissions to effect a lock on the resource. This includes checking for read and write permissions against the type of lock being
effected. If so, the lock is performed, modifying elements of the `ocb->attr` structure, and updating `ocb->attr->locklist` to reflect the new lock. This function calls `iofunc_lock()` to do the actual work.

If your resource manager calls `iofunc_lock_default()`, it must call `iofunc_close_dup_default()` and `iofunc_unblock_default()` in their respective handlers.

**Returns:**

- **EOK**  Successful completion.
- **EINVAL**  An invalid range was specified for the lock operation, or an invalid lock operation was attempted.
- **EBADF**  An attempt to perform a read lock on a write-only resource, or a write lock on a read-only resource was attempted.
- **ENOMEM**  Insufficient memory exists to allocate an internal lock structure.

**Classification:**

- QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

- `iofunc_func_init()`, `iofunc_lock()`, `iofunc_ocb_t`,
- `iofunc_time_update()`, `resmgr_attach()`, `resmgr_context_t`,
- `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer's Guide.*
Synopsis:

#include <sys/iofunc.h>

void iofunc_lock_free( iofunc_lock_list_t* lock, size_t size );

Arguments:

lock A pointer to the iofunc_lock_list_t list that you want to free.

size The amount of memory that you want to free.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The function iofunc_lock_free() frees lock structures allocated by iofunc_lock_calloc().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`iofunc_lock()`, `iofunc_lock_calloc`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_lock_ocb_default( resmgr_context_t *ctp,
                              void *reserved,
                              iofunc_ocb_t *ocb );
```

**Arguments:**

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `reserved` This argument must be NULL.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

**Library:**

- `libc`

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_lock_ocb_default()` function calls `iofunc_attr_lock()` to enforce locking on the attributes for the group of messages that were sent by the client.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `lock_ocb` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.
Returns:

EOK       Success.
EAGAIN    On the first use, all kernel mutex objects were in use.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

iofunc_attr_lock(), iofunc_func_init(), iofunc_ocb_t,
resmgr_attach(), resmgr_context_t, resmgr_io_funcs_t

Writing a Resource Manager chapter of the Programmer's Guide.
**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_lseek ( resmgr_context_t* ctp,
                   io_lseek_t* msg,
                   iofunc_ocb_t* ocb,
                   iofunc_attr_t* attr );
```

**Arguments:**

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_lseek_t` structure that contains the message that the resource manager received; see below.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `attr` A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_lseek()` helper function implements POSIX semantics for the client’s `lseek()` call, which is received as an `_IO_LSEEK` message by the resource manager.

The `iofunc_lseek()` function handles the three different `whence` cases: SEEK_SET, SEEK_CUR, and SEEK_END, updating the `ocb->offset` field with the new position.
Note that if the IOFUNC_MOUNT_32BIT flag isn’t set in the mount structure, `iofunc_lseek()` handles 64-bit position offsets. If the flag is set (meaning this device supports only 32-bit offsets), the resulting offset value is treated as a 32-bit offset, and if it overflows 32 bits, it’s truncated to LONG_MAX. Also, this function handles combine messages correctly, simplifying the work required to support lseek.

**io_lseek_t structure**

The `io_lseek_t` structure holds the _IO_LSEEK message received by the resource manager:

```c
struct _io_lseek {
    uint16_t type;
    uint16_t combine_len;
    short whence;
    uint16_t zero;
    uint64_t offset;
};

typedef union {
    struct _io_lseek i;
    uint64_t o;
} io_lseek_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The `i` member is a structure of type `io_lseek_t` that contains the following members:

- **type** — _IO_LSEEK.
- **combine_len** — If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **whence** — SEEK_SET, SEEK_CUR, or SEEK_END.
- **offset** — The relative offset from the file position determined by the `whence` member.
The $o$ member is the offset after the operation is complete.

**Returns:**

- **EOK** Successful completion.
- **EINVAL** The `whence` member in the _IO_LSEEK message wasn’t one of SEEK_SET, SEEK_CUR, or SEEK_END, or the resulting position after the offset was applied resulted in a negative number (overflow).

**Classification:**

- QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_attr_t`, `iofunc_lseek_default()`, `iofunc_ocb_t`, `lseek()`, `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
Default handler for _IO_LSEEK messages

Synopsis:

```
#include <sys/iofunc.h>

int iofunc_lseek_default( resmgr_context_t* ctp,
                          io_lseek_t* msg,
                          iofunc_ocb_t* ocb );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_lseek_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_lseek()`.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_lseek_default()` function implements POSIX semantics for the client’s `lseek()` call, which is received as an _IO_LSEEK message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `lseek` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_lseek_default()` function calls `iofunc_lseek()` to do the actual work.
**Returns:**

- **EOK**  
  Successful completion.
- **EINVAL**  
  The *whence* member in the _IO_LSEEK message wasn’t one of SEEK_SET, SEEK_CUR, or SEEK_END, or the resulting position after the offset was applied resulted in a negative number (overflow).

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_func_init()`, `iofunc_lseek()`, `iofunc_ocb_t.lseek()`, `resmgr_attach()`, `resmgr_context_t.resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
iofunc_mknod()  © 2007, QNX Software Systems GmbH & Co. KG.
Verify a client’s ability to make a new filesystem entry point

Synopsis:
#include <sys/iofunc.h>

int iofunc_mknod( resmgr_context_t *ctp,
                   io_mknod_t *msg,
                   iofunc_attr_t *attr,
                   iofunc_attr_t *dattr,
                   struct _client_info *info );

Arguments:

ctp  A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.

msg  A pointer to the io_mknod_t structure that contains the message that the resource manager received; see below.

attr NULL, or a pointer to the iofunc_attr_t structure that describes the characteristics of the resource.

dattr A pointer to the iofunc_attr_t structure that you must set. The iofunc_attr_t structure describes the attributes of the parent directory.

info NULL, or a pointer to a _client_info structure that contains information about the client. For information about this structure, see ConnectClientInfo().

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.
iofunc_mknod()

Description:

The `iofunc_mknod()` helper function supports `mknod()` requests by verifying that the client can make a new filesystem entry point. It’s similar to `iofunc_open()`.

The `iofunc_mknod()` function checks to see if the client (described by the optional `info` structure) has access to open the resource (name passed in the `msg` structure). The `attr` structure describes the resource’s attributes, and the optional `dattr` structure defines the attributes of the parent directory (i.e. if `dattr` isn’t NULL, it implies that the resource identified by `attr` is being created within the directory specified by `dattr`).

The `info` argument can be passed as NULL, in which case `iofunc_mknod()` obtains the client information itself via a call to `iofunc_client_info()`. It is, of course, more efficient to get the client info once, rather than calling this function with NULL every time.

If an error occurs, the function returns information about a client’s connection in `info` and a constant.

io_mknod_t structure

The `io_mknod_t` structure holds the _IO_CONNECT message received by the resource manager:

```
typedef union {
    struct _io_connect connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_mknod_t;
```

This message structure is a union of an input message (coming to the resource manager), _io_connect, and two possible output or reply messages (going back to the client):

- _io_connect_link_reply if the reply is redirecting the client to another resource

  Or:

- _io_connect_ftype_reply if the reply consists of a status and a file type.
Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EBADFSYS</td>
<td>NULL was passed in <code>dattr</code>.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the <code>info</code> buffer.</td>
</tr>
<tr>
<td>EINV</td>
<td>The client process is no longer valid.</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>NULL was passed in <code>info</code>.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The group ID or owner ID didn’t match.</td>
</tr>
</tbody>
</table>

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

```
_io_connect, _io_connect_link_reply,
_io_connect_fstype_reply, iofunc_client_info(), iofunc_open(),
mknod()
```

Writing a Resource Manager chapter of the *Programmer's Guide*. 
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_mmap ( resmgr_context_t * hdr,
     io_mmap_t * msg,
     iofunc_ocb_t * ocb,
     iofunc_attr_t * attr );
```

Arguments:

- `hdr`: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg`: A pointer to the `io_mmap_t` structure that contains the message that the resource manager received; see below.
- `ocb`: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `attr`: A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_mmap()` helper function provides functionality for the _IO_MMAP message. The _IO_MMAP message is an outcall from the Memory Manager (a part of the QNX Neutrino microkernel’s `procnto`).

Note that if the Process Manager is to be able to execute from this resource, then you must use the `iofunc_mmap()` function.
io_mmap_t structure

The io_mmap_t structure holds the _IO_MMAP message received by the resource manager:

```c
struct _io_mmap {
    uint16_t type;
    uint16_t combine_len;
    uint32_t prot;
    uint64_t offset;
    struct _msg_info info;
    uint32_t zero[6];
};

struct _io_mmap_reply {
    uint32_t zero;
    uint32_t flags;
    uint64_t offset;
    int32_t coid;
    int32_t fd;
};

typedef union {
    struct _io_mmap i;
    struct _io_mmap_reply o;
} io_mmap_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The i member is a structure of type _io_mmap that contains the following members:

- **type** _IO_MMAP.
- **combine_len** If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **prot** The access capabilities that the client wants to use for the memory region being mapped. This can be a combination of at least the following protection bits, as defined in `<sys/mman.h>`:
• PROT_EXEC — the region can be executed.
• PROT_NOCACHE — disable caching of the region (e.g., so it can be used to access dual-ported memory).
• PROT_NONE — the region can’t be accessed.
• PROT_READ — the region can be read.
• PROT_WRITE — the region can be written.

offset The offset into shared memory of the location that the client wants to start mapping.

info A pointer to a _msg_info, structure that contains information about the message received by the resource manager.

The o member of the io_mmap_t structure is a structure of type _io_mmap_reply that contains the following members:

flags Reserved for future use.
offset Reserved for future use.
coid A file descriptor that the process manager can use to access the mapped file.
fd Reserved for future use.

Returns:
A nonpositive value (i.e., \( \leq 0 \))

Successful completion.

EROFS An attempt to memory map (mmap) a read-only file, using the PROT_WRITE page protection mode.

EACCES The client doesn’t have the appropriate permissions.

ENOMEM Insufficient memory exists to allocate internal resources required to effect the mapping.
Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_attr_t, iofunc_mmap_default(), iofunc_ocb_t, _msg_info, resmgr_context_t

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**iofunc mmap_default()**

Default handler for _IO_MMAP messages

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_mmap_default ( resmgr_context_t * hdr,
                          io_mmap_t * msg,
                          iofunc_ocb_t * ocb );
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg** A pointer to the `io_mmap_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_mmap()`.
- **ocb** A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

**Library:**

- **libc**

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_mmap_default()` function provides functionality for the _IO_MMAP message. This message is private to the Memory Manager (a part of the Neutrino microkernel's procnto).

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `mmap` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

Note that if the Process Manager is to be able to execute from this resource, then you must use the `iofunc_mmap()` function.
The `iofunc_mmap_default()` function calls `iofunc_mmap()` to do the actual work.

**Returns:**

A nonpositive value (i.e. ≤ 0)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EROFS</td>
<td>An attempt to memory map (mmap) a read-only file, using the PROT_WRITE page protection mode.</td>
</tr>
<tr>
<td>EACCES</td>
<td>The client doesn’t have the appropriate permissions.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory exists to allocate internal resources required to effect the mapping.</td>
</tr>
</tbody>
</table>

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`iofunc_func_init()`, `iofunc_mmap()`, `iofunc_ocb_t`, `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

---

1296 C Library — I to L September 10, 2007
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_notify( resmgr_context_t *ctp,
    io_notify_t *msg,
    iofunc_notify_t *nop,
    int trig,
    const int *notifycounts,
    int *armed );
```

Arguments:

- **ctp**
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg**
  A pointer to the `io_notify_t` structure that contains the message that the resource manager received; see below.

- **nop**
  An array of `iofunc_notify_t` structures that represent the events supported by the calling resource manager. Traditionally this array contained three members which represent, in order, the input, output, and out-of-band notification lists. Since the addition of extended events (see below), three is now the minimum size of this array. The actual size must support indexing by the conditions being triggered up to `_NOTIFY_MAXCOND`.

  Generally, this structure is maintained by the resource manager within an extended attributes structure.

- **trig**
  A bitmask indicating which sources are currently satisfied, and could cause a trigger to occur. This bitmask may be indicated via two sets of flags. Traditionally, the value was any combination of `_NOTIFY_COND_INPUT`, `_NOTIFY_COND_OUTPUT` and
_NOTIFY_COND_OBAND. With the addition of extended events, this can also be any combination of the _NOTIFY_CONDE* flags. Note the following flags are considered equivalent:

\[
\begin{align*}
&\_\text{NOTIFY\_COND\_INPUT} == \_\text{NOTIFY\_CONDE\_RDNORM} \\
&\_\text{NOTIFY\_COND\_OUTPUT} == \_\text{NOTIFY\_CONDE\_WRNORM} \\
&\_\text{NOTIFY\_COND\_OBAND} == \_\text{NOTIFY\_CONDE\_RDBAND}
\end{align*}
\]

Setting the _NOTIFY_COND_EXTEN flag affects the \textit{armed} parameter, as described below.

You typically set this value, based on the conditions in effect at the time of the call.

**notifycounts**  
NULL, or an array of integers representing the number of elements that must be present in the queue of each event represented by the \textit{nop} array in order for the event to be triggered. Both this array and the \textit{nop} array should contain the same number of elements. Note that if any condition is met, nothing is armed. Only if none of the conditions are met, does the event get armed in accordance with the \textit{notifycounts} parameter. If this parameter is NULL, a value of 1 is assumed for all counts.

**armed**  
NULL, or a pointer to a location where the function can store a 1 to indicate that a notification entry is armed, or a 0 otherwise.

If you set the _NOTIFY_COND_EXTEN bit in the \textit{trig} argument, and \textit{armed} isn’t NULL, then when you call \textit{iofunc\_notify()}, \textit{armed} must contain the number of elements in the \textit{nop} and \textit{notifycounts} arrays (provided \textit{notifycounts} isn’t NULL). Otherwise, the function assumes there are three elements in the \textit{nop} and \textit{notifycounts} arrays.
Library:

 libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The POSIX layer helper function `iofunc_notify()` is used by a resource manager to implement notification.

This routine examines the message that the resource manager received (passed in the `msg` argument), and determines what action the client code is attempting to perform:

__NOTIFY_ACTION_POLL

Return a one-part IOV with the `flags` field set to indicate which conditions (input, output, or out-of-band) are available. The caller should return (__RESMGR_NPARTS(1)) to the resource manager library, which returns a one-part message to the client.

__NOTIFY_ACTION_POLLLARM

Similar to __NOTIFY_ACTION_POLL, with the additional characteristic of arming the event if none of the conditions is met.

__NOTIFY_ACTION_TRANARM

For each of the sources specified, create a notification entry and store the client’s `struct sigevent` event structure in it. Note that only one transition arm is allowed at a time per device. If the client specifies an event of SIGEV_NONE, the action is to disarm. When the event is triggered, the notification is automatically disarmed.

`io_notify_t` structure

The `io_notify_t` structure holds the __IO_NOTIFY message received by the resource manager:
struct _io_notify {
    uint16_t type;
    uint16_t combine_len;
    int32_t action;
    int32_t flags;
    struct sigevent event;
};

struct _io_notify_reply {
    uint32_t zero;
    uint32_t flags;
};

typedef union {
    struct _io_notify i;
    struct _io_notify_reply o;
} io_notify_t;

The I/O message structures are unions of an input message (coming to
the resource manager) and an output or reply message (going back to
the client).

The i member is a structure of type _io_notify that contains the
following members:

type _IO_NOTIFY.

combine_len If the message is a combine message,
_IO_COMBINE_FLAG is set in this member. For
more information, see “Combine messages” in the
Writing a Resource Manager chapter of the
Programmer’s Guide.

action _NOTIFY_ACTION_POLL,
      _NOTIFY_ACTION_POLLARM, or
      _NOTIFY_ACTION_TRANARM, as described
      above.

flags One of the following:

• _NOTIFY_COND_INPUT — this condition is
  met when there are one or more units of input
data available (i.e. clients can now issue reads).
iofunc_notify()

- _NOTIFY_COND_OUTPUT — this condition is met when there’s room in the output buffer for one or more units of data (i.e. clients can now issue writes).
- _NOTIFY_COND_OBAND — the condition is met when one or more units of out-of-band data are available.

`event` A pointer to a `sigevent` structure that defines the event that the resource manager is to deliver once a condition is met.

The `o` member is a structure of type `_io_notify_reply` that contains the following members:

`flags` Which of the conditions were triggered; see the `flags` for `iofunc_notify`, above.

**iofunc_notify_t structure**

The `iofunc_notify_t` structure is defined in `<sys/iofunc.h>` as follows:

```c
typedef struct _iofunc_notify {
    int cnt;
    struct _iofunc_notify_event *list;
} iofunc_notify_t;
```

The members of the `iofunc_notify_t` structure include:

`cnt` The smallest `cnt` member in the list; see below.

`list` A pointer to a linked list of `iofunc_notify_event_t` structures that represent (in order), the input, output, and out-of-band notification lists.

The `iofunc_notify_event_t` structure is defined as:
typedef struct _iofunc_notify_event {
    struct _iofunc_notify_event *next;
    int rcvid;
    int scoid;
    int cnt;
    struct sigevent event;
} iofunc_notify_event_t;

The members of the iofunc_notify_event_t structure include:

next A pointer to the next element in the list.
rcvid The receive ID of the client to notify.
scoid The server connection ID.
cnt The number of bytes available. Some clients, such as io-char, may want a sufficiently large amount of data to be available before they access it.
event A pointer to a sigevent structure that defines the event that the resource manager is to deliver once a condition is met.

Returns:

EBUSY A notification was already armed for this resource, and this library function enforces a restriction of one per resource.

_RESMGR_NPARTS (1)
Normal return, indicates a one-part IOV should be returned to the client.

Examples:

See the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide.
iofunc_notify()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_notify_remove(), iofunc_notify_trigger(), _RESMGR_NPARTS(), sigevent

“Handling ionotify() and select()” in the Writing a Resource Manager chapter of the Neutrino Programmer’s Guide
iofunc_notify_remove()

Remove notification entries from list

Synopsis:

```c
#include <sys/iofunc.h>

void iofunc_notify_remove( resmgr_context_t * ctp,
                          iofunc_notify_t * nop );
```

Arguments:

- `ctp` NULL, or a pointer to a `resmgr_context_t` structure for the client whose entries you want to remove.
- `nop` An array of three `iofunc_notify_t` structures that represent (in order), the input, output, and out-of-band notification lists whose entries you want to remove; for information about this structure, see the documentation for `iofunc_notify()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_notify_remove()` function removes all of the entries associated with the current client from the notification list passed in `nop`. The client information is obtained from the `ctp`.

If the `ctp` pointer is NULL, then all of the notify entries will be removed. A resource manager generally calls this function, with NULL as the `ctp` in the `close_ocb` callout, to clean up all handles associated with this connection. If the handles are shared between several connections, then the `ctp` should be provided to clean up after each client.
Examples:

See the “Writing a Resource Manager” chapter in the *Programmer’s Guide*.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`iofunc_notify()`, `iofunc_notify_trigger()`

“Handling `ionotify()` and `select()`” in the Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**iofunc_notify_trigger()**

Send notifications to queued clients

**Synopsis:**
```
#include <sys/iofunc.h>

void iofunc_notify_trigger( iofunc_notify_t *nop,
    int count,
    int index );
```

**Arguments:**
- `nop` An array of three `iofunc_notify_t` structures that represent (in order), the input, output, and out-of-band notification lists whose entries you want to examine; for information about this structure, see the documentation for `iofunc_notify()`.
- `count` The count that you want to compare to the trigger value for the event.
- `index` The index into the `nop` array that you want to check; one of the following:
  - `IOFUNC_NOTIFY_INPUT`
  - `IOFUNC_NOTIFY_OUTPUT`
  - `IOFUNC_NOTIFY_OBAND`

**Library:**
`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `iofunc_notify_trigger()` function examines all entries given in the list maintained at `nop [index]` to see if the event should be delivered to the client. If the specified `count` is greater than the trigger count for the particular notification list element, this function calls `MsgDeliverEvent()` to deliver the event to the client whose `rcvid` is stored in the notification list element, and the list element is disarmed.
Note that if the client has specified a code of SI_NOTIFY, then the value that the client specified (e.g. the value member of the struct sigevent) has the top three bits ORed with the reason for the trigger (this is the expression NOTIFY_COND_INPUT $\ll$ index), as in the following table:

\[
\begin{align*}
\text{index} &= \text{IOFUNC_NOTIFY_INPUT} \\
&= 0x10000000, \text{ or } \text{NOTIFY_COND_INPUT} \\
\text{index} &= \text{IOFUNC_NOTIFY_OUTPUT} \\
&= 0x20000000, \text{ or } \text{NOTIFY_COND_OUTPUT} \\
\text{index} &= \text{IOFUNC_NOTIFY_OBAND} \\
&= 0x40000000, \text{ or } \text{NOTIFY_COND_OBAND}
\end{align*}
\]

If the client has specified a code of something other than SI_NOTIFY then this routine doesn’t modify the value member in any way.

**Examples:**

See the Writing a Resource Manager chapter of *Programmer’s Guide*.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`iofunc_notify()`, `iofunc_notify_remove()`, `sigevent`

“Handling `ionotify()` and `select()`” in the Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```
#include <sys/iofunc.h>

int iofunc_ocb_attach(
    resmgr_context_t * ctp,
    io_open_t * msg,
    iofunc_ocb_t * ocb,
    iofunc_attr_t * attr,
    const resmgr_io_funcs_t * io_funcs );
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg**: A pointer to the `io_open_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_open()`.

- **ocb**: NULL, or a pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

- **attr**: A pointer to a `iofunc_attr_t` structure that defines the characteristics of the device that the resource manager handles.

- **io_funcs**: A pointer to a `resmgr_io_funcs_t` that specifies the I/O functions for the resource manager.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `iofunc_ocb_attach()` function examines the mode specified by the `io_open` msg, and increments the read and write count flags (`ocb->attr->rcount` and `ocb->attr->wcount`), and the locking flags (`ocb->attr->rlocks` and `ocb->attr->wlocks`), as specified by the open mode.

This function is called by `iofunc_open_default()` as part of its initialization.

This function allocates the memory for the OCB if you pass NULL as the `ocb`.

Returns:

EOK Successful completion.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`iofunc_attr_init()`, `iofunc_attr`, `iofunc_ocb_detach()`, `iofunc_ocb_t`, `iofunc_open_default()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer's Guide*. 
Synopsis:

```c
#include <sys/iofunc.h>

iofunc_ocb_t * iofunc_ocbcalloc(
    resmgr_context_t * ctp,
    iofunc_attr_t * attr);
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **attr**: A pointer to a `iofunc_attr_t` structure that defines the characteristics of the device that the resource manager handles.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_ocb calloc()` function allocates an iofunc OCB. It has a number of uses:

- It can be used as a helper function to encapsulate the allocation of the iofunc OCB, so that your routines don’t have to know the details of the iofunc OCB structure.

- Because it’s in the resource manager shared library, you can override this function with your own, allowing you to manage an OCB that has additional members, perhaps specific to your particular resource manager. If you do this, be sure to place the iofunc OCB structure as the first element of your extended OCB, and also override the `iofunc_ocb free()` function to release memory.
Another reason to override `iofunc_ocbcalloc()` might be to place limits on the number of OCBs that are in existence at any one time; the current function simply allocates OCBs until the free store is exhausted.

You should fill in the attribute’s mount structure (i.e. the `attr->mount` pointer) instead of replacing this function.

If you specify `iofunc_ocbcalloc()` and `iofunc_ocbfree()` callouts in the attribute’s mount structure, then you should use the callouts instead of calling the standard `iofunc_ocbcalloc()` and `iofunc_ocbfree()` functions.

**Returns:**

A pointer to an `iofunc_ocb_t` OCB structure.

**Examples:**

Override `iofunc_ocbcalloc()` and `iofunc_ocbfree()` to manage an extended OCB:

```c
typedef struct
{
    iofunc_ocb_t iofuncOCB; /* the OCB used by iofunc_* */
    int myFlags;
    char moreOfMyStuff;
} MyOCBT;

MyOCBT *iofunc_ocbcalloc (resmgr_context_t *ctp,
                            iofunc_attr_t *attr)
{
    return ((MyOCBT *) calloc (1, sizeof (MyOCBT)));
}

void iofunc_ocbfree (MyOCBT *ocb)
{
    free (ocb);
}
```
iofunc_ocb_calloc()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_ocb_free(), iofunc_ocb_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Release Open Control Block resources

iofunc.ocb_detach()

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_ocb_detach( resmgr_context_t * ctp,
                        iofunc_ocb_t * ocb );
```

Arguments:

- **ctp**  A pointer to a `resmgr_context_t` structure that the
  resource-manager library uses to pass context information
  between functions.

- **ocb**  A pointer to the `iofunc_ocb_t` structure for the Open
  Control Block that was created when the client opened the
  resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is
usually included automatically.

Description:

The `iofunc_ocb_detach()` function releases any resources allocated to
the passed `ocb`, such as any memory map (mmap) entries.

This function doesn’t free the memory associated with the OCB itself.

The `iofunc_ocb_detach()` function also updates the time structure, by
calling `iofunc_time_update()`, and decrements the read, write, lock,
and use counters, according to the mode that was used to open the
resource (from `ocb->ioflag`).

The counters are incremented in `iofunc_ocb_attach()`, and represent
the number of OCBs that are using the managed resource in the
respective manners (e.g.: `ocb->attr->rcount` keeps count of how many
OCBs are using the resource specified by `attr` for read access).
If you’re are using `iofunc_mmap()` or `iofunc_mmap_default()`, you must call `iofunc_ocb_detach()` to clean up. This function is called by `iofunc_close_ocb()`.

**Returns:**

A bitwise OR of flags describing the state of the managed resource:

- **IOFUNC_OCB_LAST_READER**
  This OCB was the last one performing read operations on the resource. This flag is set when the `ocb->attr->rcount` flag is decremented to zero.

- **IOFUNC_OCB_LAST_WRITER**
  This OCB was the last one performing write operations on the resource. This flag is set when the `ocb->attr->wcount` flag is decremented to zero.

- **IOFUNC_OCB_LAST_RDLOCK**
  This OCB was the last one holding a read lock on the resource. This flag is set when the `ocb->attr->rlocks` flag is decremented to zero.

- **IOFUNC_OCB_LAST_WRLOCK**
  This OCB was the last one holding a write lock on the resource. This flag is set when the `ocb->attr->wlocks` flag is decremented to zero.

- **IOFUNC_OCB_LAST_INUSE**
  This OCB was the last one using the resource. This flag is set when the `ocb->attr->count` flag is decremented to zero.

**Classification:**

QNX Neutrino
iofunc_ocb_detach()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_close_ocb(), iofunc_close_ocb_default(), iofunc_mmap(),
iofunc_mmap_default(), iofunc_ocb_attach(), iofunc_ocb_t,
iofunc_time_update(), resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Deallocate an iofunc Open Control Block’s memory

Synopsis:

```c
#include <sys/iofunc.h>

void iofunc_ocb_free( iofunc_ocb_t * ocb );
```

Arguments:

- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_ocb_free()` function returns the memory allocated to an iofunc OCB to the free store pool. This function is the complement of `iofunc_ocb_calloc()`.

If you’ve overridden the definition of `iofunc_ocb_calloc()`, you should also override the definition of `iofunc_ocb_free()` to correctly handle the release of the memory. This is because the `iofunc_ocb_calloc()` functions uses an internal memory management function to allocate the memory, and the default `iofunc_ocb_free()` function also uses this internal function to deallocate memory. Therefore, you can’t mix internal memory management functions (`_scalloc()` and `_sfree()`) with user-level memory management functions (`calloc()` and `free()`).
You should fill in the attribute’s mount structure (i.e. the `attr->mount` pointer) instead of replacing this function.

If you specify `iofunc_ocb_free()` and `iofunc_ocb_calloc()` callouts in the attribute’s mount structure, then you should use the callouts instead of calling the standard `iofunc_ocb_free()` and `iofunc_ocb_calloc()` functions.

Examples:

See `iofunc_ocb_calloc()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`iofunc_close_ocb()`, `iofunc_ocb_calloc()`, `iofunc_ocb_t`  
Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```
#include <sys/iofunc.h>

typedef struct _iofunc_ocb {
    IOFUNC_ATTR_T *attr;
    int32_t ioflag;
#if !defined(_IOFUNC_OFFSET_BITS) || _IOFUNC_OFFSET_BITS == 64
    #if _FILE_OFFSET_BITS - 0 == 64
        off_t offset;
    #else
        off64_t offset;
    #endif
#elif _IOFUNC_OFFSET_BITS - 0 == 32
    #if !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
        #if defined(__LITTLEENDIAN__)
            off_t offset;
            off_t offset_hi;
        #elif defined(__BIGENDIAN__)
            off_t offset_hi;
            off_t offset;
        #else
            #error endian not configured for system
        #endif
    #else
        #if defined(__LITTLEENDIAN__)
            int32_t offset;
            int32_t offset_hi;
        #elif defined(__BIGENDIAN__)
            int32_t offset_hi;
            int32_t offset;
        #else
            #error endian not configured for system
        #endif
    #endif
#else
    #error _IOFUNC_OFFSET_BITS value is unsupported
#endif
    uint16_t sflag;
    uint16_t flags;
    void *reserved;
} iofunc_ocb_t;
```
Description:

The `iofunc_ocb_t` structure is an Open Control Block, a block of data that’s established by a resource manager during its handling of the client’s `open()` function.

A resource manager creates an instance of this structure whenever a client opens a resource. For example, `iofunc_open_default()` calls `iofunc_ocb_calloc()` to allocate an OCB. The OCB exists until the client closes the file descriptor associated with the open operation. The resource manager passes this structure to all of the functions that implement the I/O operations for the file descriptor.

The `iofunc_ocb_t` structure includes the following members:

- `attr` A pointer to the OCB’s attributes. By default, this structure is of type `iofunc_attr_t`, but you can redefine the IOFUNC_ATTR_T manifest if you want to use a different structure in your resource manager.

- `ioflag` The mode (e.g. reading, writing, blocking) that the resource was opened with.

The bits in this member are the same as those for the `oflag` argument to `open()` plus 1.

This information is inherited from the `io_connect_t` structure that’s available in the message passed to the open handler.

- `offset, offset_hi` The read/write offset into the resource (e.g. our current `lseek()` position within a file), defined in a variety of ways to suit 32- and 64-bit offsets. Your resource manager can modify this offset.

- `sflag` The sharing mode; see `sopen()`. This information is inherited from the `io_connect_t` structure that’s available in the message passed to the open handler.
flags When the IOFUNC_OCB_PRIVILEGED bit is set, a
privileged process (i.e. root) performed the open().
Additionally, you can use flags in the range defined by
IOFUNC_OCB_FLAGS_PRIVATE (see <sys/iofunc.h>)
for your own purposes. Your resource manager can modify
these flags.

Classification:

QNX Neutrino

See also:

iofunc_attr_t, iofunc_ocb_calloc(), iofunc_open_default()

Writing a Resource Manager chapter of the Programmer’s Guide.
Verify a client’s ability to open a resource

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_open( resmgr_context_t *ctp,
                 io_open_t *msg,
                 iofunc_attr_t *attr,
                 iofunc_attr_t *dattr,
                 struct _client_info *info );
```

Arguments:

- `ctp`: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg`: A pointer to the `io_open_t` structure that contains the message that the resource manager received; see below.
- `attr`: A pointer to the `iofunc_attr_t` structure that describes the characteristics of the resource.
- `dattr`: NULL, or a pointer to the `iofunc_attr_t` structure that describes the characteristics of the parent directory.
- `info`: NULL, or a pointer to a `_client_info` structure that contains the information about a client connection. For information about this structure, see `ConnectClientInfo()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_open()` function checks to see if the client (described by the optional `info` structure) has access to open the resource whose name is passed in `msg->connect.path`. 
The `attr` structure describes the resource’s attributes. The optional `dattr` structure defines the attributes of the parent directory; if `dattr` isn’t `NULL`, the resource identified by `attr` is being created within the directory specified by `dattr`.

The `info` argument can be passed as `NULL`, in which case `iofunc_open()` obtains the client information itself via a call to `iofunc_client_info()`. It is, of course, more efficient to get the client info once, rather than calling this function with `NULL` every time.

Note that if you’re handling a request to read directory entry, you must return data formatted to match the `struct dirent` type. A helper function, `iofunc_stat()`, can aid in this.

A resource manager’s response to an `open()` request isn’t always a yes-or-no answer. It’s possible to return a connect message indicating that the server would like some other action taken. For example, if the open occurs on a path that represents a symbolic link to some other path, the server could respond using the `_IO_SET_CONNECT_RET()` macro and the `_IO_CONNECT_RET_LINK` value.

For example, an open handler that only redirects pathnames might look something like:

```c
io_open(resmgr_context_t *ctp, io_open_t *msg, 
    iofunc_attr_t *dattr, void *extra) {
    char *newpath;
    
    /* Do all the error/access checking ... */
    /* Lookup the redirected path and store 
    the new path in ‘newpath’ */
    newpath = get_a_new_path(msg->connect.path);

    _IO_SET_CONNECT_RET(ctp, _IO_CONNECT_RET_LINK);
    len = strlen(newpath) + 1;

    msg->link_reply.eflag = msg->connect.eflag;
    msg->link_reply.nentries = 0;
    msg->link_reply.path_len = len;
    strcpy((char *)(msg->link_reply + 1), newpath);
    len += sizeof(msg->link_reply);
```
In this example, we use the macro \_IO\_SET\_CONNECT\_RET() (defined in </sys/iomsg.h>) to set the ctp->status field to _IO\_CONNECT\_RET\_LINK. This value indicates to the resource-manager framework that the return value isn’t actually a simple return code, but a new request to be processed.

The path for this new request follows directly after the link_reply structure and is path_len bytes long. The final few lines of the code just stuff an IOV with the reply message (and the new path to be queried) and return to the resource-manager framework.

**io\_open_t structure**

The io\_open\_t structure holds the _IO\_CONNECT message received by the resource manager:

```c
typedef union {
    struct _io_connect connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_open_t;
```

This message structure is a union of an input message (coming to the resource manager), _io_connect, and two possible output or reply messages (going back to the client):

- _io_connect_link_reply if the reply is redirecting the client to another resource
  
  Or:

- _io_connect_ftype_reply if the reply consists of a status and a file type.

**Returns:**

- EOK Successful completion.
- Other There was an error, as defined by the POSIX semantics for the open call. This error should be returned to the next higher level.
Examples:

This is a sample skeleton for a typical filesystem, in pseudo-code, to illustrate the steps that need to be taken to handle an open request for a file:

```c
if the open request is for a path (i.e. multiple directory levels)
    call iofunc_client_info to get information about client
    for each directory component
        call iofunc_check_access to check execute permission for access
        /*
          recall that execute permission on a directory is really the "search" permission for that directory
        */
        next
    /*
      at this point you have verified access to the target
    */
endif

if O_CREAT is set and the file doesn't exist
    call iofunc_open, passing the attribute of the parent as dattr
    if the iofunc_open succeeds,
        do the work to create the new inode, or whatever
    endif
else
    call iofunc_open, passing the attr of the file and NULL for dattr
endif

/*
at this point, check for things like o_trunc, etc. -- things that you have to do for the attr
*/

call iofunc_ocb_attach
return EOK
```

For a device (i.e. resmgr_attach() didn’t specify that the managed resource is a directory), the following steps apply:

```c
/*
```
at startup time (i.e.: in the main() of the 
resource manager)
*/
call iofunc_attr_init to initialize an attribute 
structure

/* in the io_open message handler: */
call iofunc_open, passing in the attribute of the 
device and NULL for dattr

call iofunc_ocb_attach
return EOK

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

_io_connect, _io_connect_link_reply, 
_io_connect_ftype_reply, iofunc_attr_init(), 
iofunc_check_access(), iofunc_client_info(), iofunc_ocb_attach(), 
iofunc_stat(), resmgr_open_bind()

Writing a Resource Manager chapter of the *Programmer's Guide*. 
Synopsis:
#include <sys/iofunc.h>

int iofunc_open_default( resmgr_context_t *ctp,
                        io_open_t *msg,
                        iofunc_attr_t *attr,
                        void *extra );

Arguments:

ctp A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.

msg A pointer to the io_open_t structure that contains the message that the resource manager received. For more information, see the documentation for iofunc_open().

attr A pointer to the iofunc_attr_t structure that defines the characteristics of the device that the resource manager is controlling.

extra Extra information from the library. If you’re calling iofunc_open_default() from a resource manager’s open() function (see resmgr_connect_funcs_t), simply pass the extra argument that’s passed to open().

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The iofunc_open_default() function implements the default actions for the _IO_CONNECT message in a resource manager. This function calls:
iofunc_open_default()

- `iofunc_open()` to check the client’s open mode against the resources attributes to see if the client can open the resource in that mode
- `iofunc_ocbcalloc()` to allocate an Open Control Block (OCB)
- `iofunc_ocbattach()` to initialize the OCB
- `resmgr_openbind()` to bind the newly-created OCB to the request.

You can place this function directly into the `connect_funcs` table passed to `resmgr_attach()`, at the `open` position, or you can call `iofunc_funcinit()` to initialize all of the functions to their default values.

See the “Examples” section in the description of `iofunc_open()` for the skeleton outline of the functionality (the second example, where `resmgr_attach()` doesn’t specify that the managed resource is a directory).

Returns:

- **EOK** Successful completion.
- **ENOSPC** There’s insufficient memory to allocate the OCB.
- **ENOMEM** There’s insufficient memory to allocate an internal data structure required by `resmgr_openbind()`.

Classification:

QNX Neutrino

Safety

- Cancellation point: No
- Interrupt handler: No

continued...
See also:

iofunc_func_init(), iofunc_ocb_attach(), iofunc_ocb_calloc(),
iofunc_open(), iofunc_time_update(), resmgr_attach(),
resmgr_connect_funcs_t, resmgr_open_bind()

Writing a Resource Manager chapter of the Programmer’s Guide.
#Synopsis:

```
#include <sys/iofunc.h>

int iofunc_openfd( resmgr_context_t *ctp,
                    io_openfd_t *msg,
                    iofunc_ocb_t *ocb,
                    iofunc_attr_t *attr );
```

##Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_openfd_t` structure that contains the message that the resource manager received; see below.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `attr` A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

##Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

##Description:

The `iofunc_openfd()` helper function examines the mode specified by the `IO_OPENFD` message, and increments the read and write count flags (`ocb->attr->rcount` and `ocb->attr->wcount`), and the locking flags (`ocb->attr->rlocks` and `ocb->attr->wlocks`), as specified by the open mode.

The function does what’s needed to support the `openfd()` function.
io_openfd_t structure

The io_openfd_t structure holds the _IO_OPENFD message received by the resource manager:

```c
struct _io_openfd {
    uint16_t type;
    uint16_t combine_len;
    uint32_t ioflag;
    uint16_t sflag;
    uint16_t xtype;
    struct _msg_info info;
    uint32_t reserved2;
    uint32_t key;
};
```

typedef union {
    struct _io_openfd i;
} io_openfd_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, i.

The i member is a structure of type _io_openfd that contains the following members:

- **type** — _IO_OPENFD.
- **combine_len** — If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **ioflag** — How the client wants to open the file; a combination of the following bits:
  - O_RDONLY — permit the file to be only read.
  - O_WRONLY — permit the file to be only written.
  - O_RDWR — permit the file to be both read and written.
iofunc_openfd()

- **O_APPEND** — cause each record that’s written to be written at the end of the file.
- **O_TRUNC** — if the file exists, truncate it to contain no data. This flag has no effect if the file doesn’t exist.

**sflag**

How the client wants the file to be shared; a combination of the following bits:

- **SH_COMPAT** — set compatibility mode.
- **SH_DENYRW** — prevent read or write access to the file.
- **SH_DENYWR** — prevent write access to the file.
- **SH_DENYRD** — prevent read access to the file.
- **SH_DENYNO** — permit both read and write access to the file.

**xtype**

Extended type information that can change the behavior of an I/O function. One of:

- **_IO_OPENFD_NONE** — no extended type information.
- **_IO_OPENFD_PIPE** — a pipe is being opened.
- **_IO_OPENFD_RESERVED** — reserved

**info**

A pointer to a `_msg_info` structure that contains information about the message received by the resource manager.

**key**

Reserved for future use.

**Returns:**

- **EOK** Success.
- **EACCES** You don’t have permission to open the file.
- **EBUSY** The file has shared locks that are in use.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`iofunc_openfd()`, `iofunc_openfd_default()`, `_msg_info.openfd()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_openfd_default( resmgr_context_t *ctp,
                          io_openfd_t *msg,
                          iofunc_ocb_t *ocb );
```

**Arguments:**

- **ctp**  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg**  A pointer to the `io_openfd_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_openfd()`.

- **ocb**  A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The function `iofunc_openfd_default()` function implements the default actions for _IO_OPENFD messages.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `openfd` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_openfd_default()` function calls `iofunc_openfd()` to do the actual work.
Returns:

- **EOK**: Success.
- **EACCES**: You don’t have permission to open the file.
- **EBUSY**: The file has shared locks that are in use.
- **EINVAL**: The message type is invalid.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_chown_default()`, `iofunc_func_init()`, `iofunc_ocb_t`,
- `iofunc_openfd()`, `iofunc_sync_default()`, `resmgr_attach()`,
- `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

September 10, 2007
iofunc_pathconf()  
Support pathconf() requests

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_pathconf( resmgr_context_t *ctp, 
                      io_pathconf_t *msg, 
                      iofunc_ocb_t *ocb, 
                      iofunc_attr_t *attr );
```

Arguments:

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg** A pointer to the `io_pathconf_t` structure that contains the message that the resource manager received; see below.

- **ocb** A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

- **attr** A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that's associated with your resource manager.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_pathconf()` helper function does what’s needed to support `pathconf()` with the `mount` and `attr` passed to it. Other `fsys pathconf()` requests need to be handled by the caller.

If you write your own `pathconf` callout for your resource manager, use the following macro to pass the requested value back to the caller:
_IO_SET_PATHCONF_VALUE( resmgr_context_t *ctp,  
    int value )

io_pathconf_t structure

The io_pathconf_t structure holds the _IO_PATHCONF message received by the resource manager:

```c
struct _io_pathconf {
    uint16_t   type;  
    uint16x_t  combine_len;  
    short      name;  
    uint16_t   zero;  
};
typedef union {
    struct _io_pathconf i;  
    /* value is returned with MsgReply */
} io_pathconf_t;
```

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, i.

The i member is a structure of type _io_pathconf that contains the following members:

- `type`  
  _IO_PATHCONF.

- `combine_len`  
  If the message is a combine message,  
  _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.

- `name`  
  The name of the configurable limit; see pathconf().

Returns:

EOK, or _RESMGR_DEFAULT if the function didn’t handle the pathconf() request.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_attr_t, iofunc_ocb_t, iofunc_pathconf_default(), pathconf(), resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_pathconf_default( resmgr_context_t *ctp,
                               io_pathconf_t *msg,
                               iofunc_ocb_t *ocb );
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg**: A pointer to the `io_pathconf_t` structure that contains the message that the resource manager received. For more information, see the documentation for `iofunc_pathconf()`.
- **ocb**: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_pathconf_default()` function implements POSIX semantics for the client’s `pathconf()` call, which is received as an _IO_PATHCONF message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `pathconf` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_pathconf_default()` function returns information about the resource, as per the POSIX specifications for `pathconf()`.
`iofunc_pathconf_default()` function simply calls `iofunc_pathconf()` to do the actual work.

**Returns:**

- **EOK** Successful completion.
- **EINV** The pathconf parameter being ascertained wasn’t one of `_PC_CHOWN_RESTRICTED`, `_PC_NO_TRUNC`, or `_PC_SYNC_IO`.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_func_init()`, `iofunc_ocb_t`, `iofunc_pathconf()`, `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
ioproc_read_default()

Default handler for _IO_READ messages

Synopsis:

```c
#include <sys/ioproc.h>

int ioproc_read_default( resmgr_context_t *ctp,
                         io_read_t *msg,
                         ioproc_ocb_t *ocb);
```

Arguments:

- `ctp`: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg`: A pointer to the `io_read_t` structure that contains the message that the resource manager received. For more information, see the documentation for `ioproc_read_verify()`.
- `ocb`: A pointer to the `ioproc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ioproc_read_default()` function implements POSIX semantics for the client’s `read()` call, which is received as an _IO_READ message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `read` position, or you can call `ioproc_func_init()` to initialize all of the functions to their default values.

The `ioproc_read_default()` function calls `ioproc_read_verify()` to do the actual work.
Returns:

- **EBADF**: The client doesn’t have read access to this resource.
- **EINVAL**: The extended type information is invalid.
- **EOK**: The client has read access to this resource.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_func_init()`, `iofunc_ocb_t`, `iofunc_read_verify()`,
- `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**iofunc_read_verify()**

Verify a client’s read access to a resource

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_read_verify( resmgr_context_t* ctp,
                        io_read_t* msg,
                        iofunc_ocb_t* ocb,
                        int* nonblock );
```

**Arguments:**

- **ctp**
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg**
  A pointer to the `io_read_t` structure that contains the message that the resource manager received; see below.

- **ocb**
  A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

- **nonblock**
  NULL, or a pointer to a location where the function can store a value that indicates whether or not the device is nonblocking:
  
  - Nonzero — the client doesn’t want to be blocked (i.e. `O_NONBLOCK` was set).
  
  - Zero — the client wants to be blocked.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `iofunc_read_verify()` helper function checks that the client that sent the `_IO_READ` message actually has read access to the resource, and, if `nonblock` isn’t NULL, sets `nonblock` to `O_NONBLOCK` or 0).

The read permission check is done against `ocb->ioflag`.

Note that the `io_read_t` message has an override flag called `msg->i.xtype`. This flag allows the client to override the default blocking behavior for the resource on a per-request basis. This override flag is checked, and returned in the optional `nonblock`.

Note that if you’re reading from a directory entry, you must return `struct dirent` structures in the `read` callout for your resource manager.

You’ll also need to indicate how many bytes were read. You can do this with the macro:

```
_IOC_SET_READ_NBYTES( resmgr_context_t *ctp, int nbytes )
```

`io_read_t` structure

The `io_read_t` structure holds the `_IO_READ` message received by the resource manager:

```
struct _io_read {
    uint16_t type;
    uint16_t combine_len;
    int32_t nbytes;
    uint32_t xtype;
    uint32_t zero;
};
```

typedef union {
    struct _io_read i;
    /* unsigned char data[nbytes]; */
    /* nbytes is returned with MsgReply */
} io_read_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, `i`. 
The i member is a structure of type _io_read that contains the following members:

- **type**: _IO_READ.
- **combine_len**: If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **nbytes**: The number of bytes that the client wants to read.
- **xtype**: Extended type information; one of:
  - _IO_XTYPE_NONE
  - _IO_XTYPE_READCOND
  - _IO_XTYPE_MQUEUE
  - _IO_XTYPE_TCPIP
  - _IO_XTYPE_TCPIP_MSG
  - _IO_XTYPE_OFFSET
  - _IO_XTYPE_REGISTRY
  - _IO_XFLAG_DIR_EXTRA_HINT — this flag is valid only when reading from a directory. The filesystem should normally return extra directory information when it’s easy to get. If this flag is set, it is a hint to the filesystem to try harder (possibly causing media lookups) to return the extra information. The most common use would be to return _DTYPE_LSTAT information.
  - _IO_XFLAG_NONBLOCK
  - _IO_XFLAG_BLOCK

For more information, see “Handling other read/write details” in the Writing a Resource Manager chapter of the Programmer’s Guide.
The commented-out declaration for `data` indicates that `nbytes` bytes of data immediately follow the `io_read_t` structure.

**Returns:**

- **EOK**  The client has read access to this resource.
- **EBADF**  The client doesn’t have read access to this resource.

**Classification:**

- **QNX Neutrino**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`iofunc_open(), iofunc_write_verify()`

Writing a Resource Manager chapter of the *Programmer’s Guide*
**iofunc_readlink()**

Verify a client’s ability to read a symbolic link

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_readlink(resmgr_context_t *ctp,
                     io_readlink_t *msg,
                     iofunc_attr_t *attr,
                     struct_client_info *info);
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg** A pointer to the `io_readlink_t` structure that contains the message that the resource manager received; see below.
- **attr** A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.
- **info** A pointer to a `client_info` structure that contains the information about a client connection. For information about this structure, see `ConnectClientInfo()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_readlink()` helper function supports `readlink()` requests by verifying that the client can read a symbolic link. It’s similar to `iofunc_open()`.

The `iofunc_read()` function checks to see if the client (described by the optional `info` structure) has access to open the resource (name
passed in the msg structure). The attr structure describes the resource’s attributes.

The info argument can be passed as NULL, in which case iofunc_read() obtains the client information itself via a call to iofunc_client_info(). It is, of course, more efficient to get the client info once, rather than calling this function with NULL every time.

The iofunc_readlink() function handles the readlink verification for the POSIX layer.

io_readlink_t structure

The io_readlink_t structure holds the _IO_CONNECT message received by the resource manager:

typedef union {
    struct _io_connect connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_readlink_t;

This message structure is a union of an input message (coming to the resource manager), _io_connect, and two possible output or reply messages (going back to the client):

- _io_connect_link_reply if the reply is redirecting the client to another resource
  Or:

- _io_connect_ftype_reply if the reply consists of a status and a file type.

Returns:

EBADFSYS NULL was passed in attr.

EOK Successful completion.
Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`io_connect`, `io_connect_link_reply`,
`io_connect_ftype_reply`, `iofunc_open()`, `readlink()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**iofunc_rename()**

Do permission checks for a _IO_CONNECT_RENAME message

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_rename( resmgr_context_t* ctp,
                   io_rename_t* msg,
                   iofunc_attr_t* oldattr,
                   iofunc_attr_t* olddattr,
                   iofunc_attr_t* newattr,
                   iofunc_attr_t* newdattr,
                   struct _client_info* info );
```

**Arguments:**

- **ctp** - A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.
- **msg** - A pointer to the io_rename_t structure that contains the message that the resource manager received; see below.
- **oldattr** - A pointer to the iofunc_attr_t structure that describes the characteristics of the resource.
- **olddattr** - NULL, or a pointer to the iofunc_attr_t structure that describes the characteristics of the parent directory.
- **newattr** - A pointer to the iofunc_attr_t structure that describes the characteristics of the target, if it exists.
- **newdattr** - NULL, or a pointer to the iofunc_attr_t structure that describes the characteristics of the parent directory of the target.
- **info** - NULL, or a pointer to a _client_info structure that contains the information about a client connection. For information about this structure, see ConnectClientInfo().
iofunc_rename()

Library:

libc

Use the -l libc option to qcc to link against this library. This library is usually included automatically.

Description:

The function `iofunc_rename()` does permission checks for the `_IO_CONNECT` message (subtype `_IO_CONNECT_RENAME`) for context `ctp`. The `newattr` argument is the attribute of the target if it already exists.

This function is similar to `iofunc_open()`. The `iofunc_rename()` function checks to see if the client (described by the optional `info` structure) has access to open the resource (name passed in the `msg` structure). The `attr` structure describes the resource’s attributes.

The `info` argument can be passed as NULL, in which case `iofunc_rename()` obtains the client information itself via a call to `iofunc_client_info()`. It is, of course, more efficient to get the client information once, rather than call this function with NULL every time.

io_rename_t structure

The `io_rename_t` structure holds the `_IO_CONNECT` message received by the resource manager:

```
typedef union {
    struct _io_connect connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_rename_t;
```

This message structure is a union of an input message (coming to the resource manager), `_io_connect`, and two possible output or reply messages (going back to the client):

- `_io_connect_link_reply` if the reply is redirecting the client to another resource

Or:
- \_io\_connect\_ftype\_reply if the reply consists of a status and a file type.

The reply includes the following extra information:

```c
typedef union \_io\_rename\_extra {
    char path[1];
} io\_rename\_extra\_t;
```

**Returns:**

- EACCES  The client doesn’t have permissions to do the operation.
- EBADFSYS  NULL was passed in oldattr, oldattr, or newattr.
-EFAULT  A fault occurred when the kernel tried to access the info buffer.
- EINVAL  The oldattr and newattr have identical values, the client process is no longer valid, or attempt to remove the parent (".") directory.
- EISDIR  The old link is a directory but the new link isn’t a directory.
- ENOTDIR  Attempt to unlink a nondirectory entry using directory semantics (e.g. rmdir file).
- ENOTEMPTY  Attempt to remove a directory that isn’t empty.
- EOK  Successful completion or there was already a newattr entry.
- EPERM  The group ID or owner ID didn’t match.
- EROFS  Attempt to remove an entry on a read-only filesystem.
`iofunc_rename()`

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`_io_connect`, `_io_connect_link_reply`, `_io_connect_ftype_reply`, `iofunc_client_info()`, `iofunc_open()`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
**iofunc_space_verify()**

Do permission checks for the _IO_SPACE message_

**Synopsis:**
```
#include <sys/iofunc.h>

int iofunc_space_verify( resmgr_context_t *ctp,
                        io_space_t *msg,
                        iofunc_ocb_t *ocb,
                        int *nonblock );
```

**Arguments:**
- **ctp**
  A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.
- **msg**
  A pointer to the io_space_t structure that contains the message that the resource manager received; see below.
- **ocb**
  A pointer to the iofunc_ocb_t structure for the Open Control Block that was created when the client opened the resource.
- **nonblock**
  NULL, or a pointer to a location where the function can store a value that indicates whether or not the device is nonblocking:
  - Zero — the client doesn’t want to be blocked (i.e. O_NONBLOCK was set).
  - Nonzero — the client wants to be blocked.

**Library:**
- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The iofunc_space_verify() helper function checks the client's permission for an _IO_SPACE message.

io_space_t structure

The io_space_t structure holds the _IO_SPACE message received by the resource manager:

```c
struct _io_space {
    uint16_t type;
    uint16_t combine_len;
    uint16_t subtype;
    short whence;
    uint64_t start;
    uint64_t len;
};
```

typedef union {
    struct _io_space i;
    uint64_t o;
} io_space_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The i member is a structure of type _io_space that contains the following members:

- **type** _IO_SPACE.
- **combine_len** If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **subtype** F_ALLOCSP or F_FREESP.
- **whence** The position in the file. The possible values (defined in <unistd.h>) are:
iofunc_space_verify()

SEEK_CUR   The new file position is computed relative to the current file position. The value of start may be positive, negative or zero.

SEEK_END   The new file position is computed relative to the end of the file.

SEEK_SET   The new file position is computed relative to the start of the file. The value of start must not be negative.

start      The relative offset from the file position determined by the whence member.

len        The relative size by which to increase the file. A value of zero means to end of file.

The o member is the file size.

Returns:

EBADF      The client doesn’t have read access to this resource.
EISDIR     The resource is a directory.
EOK        The client has read access to this resource.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1356   C Library — I to L   September 10, 2007
See also:

iofunc_ocb_t, iofunc_open(), iofunc_write_default(),
iofunc_write_verify(), resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Populate a `stat` structure

**Synopsis:**
```
#include <sys/iofunc.h>

int iofunc_stat( resmgr_context_t* ctp,
                 iofunc_attr_t* attr,
                 struct stat* stat );
```

**Arguments:**
- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `attr` A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.
- `stat` A pointer to the `stat` structure that you want to fill. For more information, see `stat()`.

**Library:**
`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `iofunc_stat()` function populates the passed `stat` structure based on information from the passed `attr` structure and the context pointer, `ctp`.

This is typically used when the resource manager is handling the `IO_STAT` message, and needs to format the current status information for the resource.
Returns:

EOK Successful completion.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_attr_t, iofunc_stat_default(), iofunc_time_update(),
resmgr_context_t, stat()

Writing a Resource Manager chapter of the Programmer’s Guide.
**iofunc_stat_default()**

*Default handler for _IO_STAT messages*

**Synopsis:**

```
#include <sys/iofunc.h>

int iofunc_stat_default( resmgr_context_t *ctp,
                         io_stat_t *msg,
                         iofunc_ocb_t *ocb );
```

**Arguments:**

- **ctp**
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg**
  A pointer to the `io_stat_t` structure that contains the message that the resource manager received; see below.

- **ocb**
  A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_stat_default()` function implements POSIX semantics for the client’s `stat()` or `fstat()` call, which is received as an _IO_STAT message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `stat` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_stat_default()` function calls:

- `iofunc_time_update()`, to ensure that the time entries in the `ocb->attr` structure are current and valid.
iofunc_stat() to construct a status entry based on the information in the ocb->attr structure.

io_stat_t structure

The io_stat_t structure holds the _IO_STAT message received by the resource manager:

```c
struct _io_stat {
    uint16_t type;
    uint16_t combine_len;
    uint32_t zero;
};
```

typedef union {
    struct _io_stat i;
    struct stat o;
} io_stat_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client).

The i member is a structure of type _io_stat that contains the following members:

- **type** _IO_STAT.
- **combine_len** If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.

The o member is a structure of type stat; for more information, see stat().

**Returns:**

EOK  Successful completion.
iofunc_stat_default()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_func_init(), iofunc_ocb_t, iofunc_stat(), iofunc_time_update(), resmgr_attach(), resmgr_context_t, resmgr_io_funcs_t, stat()

Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```
#include <sys/iofunc.h>

int iofunc_sync( resmgr_context_t* ctp,  
                 iofunc_ocb_t* ocb,  
                 int ioflag );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `ioflag` The operation being performed on the OCB:
  - `_IO_FLAG_WR` — writing.
  - `_IO_FLAG_RD` — reading.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_sync()` function indicates if some form of synchronization is needed.

Returns:

- `O_DSYNC` Data integrity is needed.
- `O_SYNC` File integrity is needed.
- `0` Synchronization isn’t needed.
iofunc_sync()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_open(), iofunc_write_default(), iofunc_write_verify()

Writing a Resource Manager chapter of the Programmer’s Guide.
iofunc sync default()
Default handler for _IO_SYNC messages

Synopsis:
#include <sys/iofunc.h>

int iofunc_sync_default( resmgr_context_t *ctp,
                        io_sync_t *msg,
                        iofunc_ocb_t *ocb );

Arguments:

ctp        A pointer to a resmgr_context_t structure that the
           resource-manager library uses to pass context information
           between functions.

msg        A pointer to the io_sync_t structure that contains the
           message that the resource manager received. For more
           information, see iofunc_sync_verify().

ocb        A pointer to the iofunc_ocb_t structure for the Open
           Control Block that was created when the client opened the
           resource.

Library:
libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:
The function iofunc_sync_default() function implements POSIX
semantics for the client’s sync() call, which is received as an
_IO_SYNC message by the resource manager.

You can place this function directly into the io_funcs table passed to
resmgr_attach(), at the sync position, or you can call
iofunc_func_init() to initialize all of the functions to their default
values.

The iofunc_sync_default() function calls iofunc_sync_verify() to see
if the client can synchronize the resource.
Returns:

- **EINVAL** The resource doesn’t support synchronizing.
- **EOK** The client can synchronize the resource.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_func_init()`, `iofunc_ocb_t`, `iofunc_sync()`, `iofunc_sync_verify()`, `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:

#include <sys/iofunc.h>

int iofunc_sync_verify( resmgr_context_t *ctp,
                                         io_sync_t *msg,
                                         iofunc_ocb_t *ocb );

Arguments:

ctp      A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.

msg      A pointer to the io_sync_t structure that contains the message that the resource manager received; see below.

ocb      A pointer to the iofunc_ocb_t structure for the Open Control Block that was created when the client opened the resource.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The iofunc_sync_verify() function verifies that the client has permission to synchronize.

io_sync_t structure

The io_sync_t structure holds the _IO_SYNC message received by the resource manager:

struct _io_sync {
    uint16_t type;
    uint16_t combine_len;
    uint32_t flag;
};
typedef union {
    struct _io_sync i;
} io_sync_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there's only an input message, i.

The i member is a structure of type _io_sync that contains the following members:

- **type**: _IO_SYNC.
- **combine_len**: If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **flag**: One of:
  - O_DSYNC
  - O_RSYNC
  - O_SYNC
  For more information about these flags, see open().

**Returns:**

- EINVAL The resource doesn’t support syncing.
- EOK The client has read access to this resource.

**Classification:**

QNX Neutrino
**iofunc_sync_verify()**

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `iofunc_ocb_t`
- `iofunc_open()`, `iofunc_write_default()`, `iofunc_write_verify()`
- `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:
```
#include <sys/iofunc.h>

int iofunc_time_update( iofunc_attr_t* attr );
```

Arguments:
- `attr` A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that's associated with your resource manager.

Library:
```
lbc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `iofunc_time_update()` function examines the `flags` member in the passed `attr` structure against the bits `IOFUNC_ATTR_ATIME`, `IOFUNC_ATTR_MTIME`, and `IOFUNC_ATTR_CTIME`. If any of these bits are set, the corresponding time member of `attr` (e.g. `attr->atime`) isn’t valid. This function updates all invalid `attr` members to the current time.

If `iofunc_time_update()` makes any change to the `attr` structure’s time members, it sets `IOFUNC_ATTR_DIRTY_TIME` in the `attr` structure’s `flags` member. This function always clears the `IOFUNC_ATTR_ATIME`, `IOFUNC_ATTR_MTIME`, and `IOFUNC_ATTR_CTIME` bits from `attr->flags`.

Returns:
- `EOK` Successful completion.
iofunc_time_update()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

iofunc_attr_t

Writing a Resource Manager chapter of the Programmer’s Guide.
Unblock OCBs

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_unblock( resmgr_context_t * ctp,
                    iofunc_attr_t * attr );
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **attr** A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_unblock()` function unblocks any clients that are blocked on any internal resource manager structures. Currently, this involves only the advisory lock list that’s maintained by the attribute.

If a client connection is found:

- that client is unblocked, and is replied to with the error EINTR.
- `iofunc_unblock()` returns `_RESMGR_NOREPLY`.

If no client connection is found, `iofunc_unblock()` returns `_RESMGR_DEFAULT`. 
**iofunc_unblock()**

**Returns:**

- `_RESMGR_DEFAULT`
  
  No client connection was found.
- `_RESMGR_NOREPLY`
  
  A client connection has been unblocked.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`iofunc_unblock_default()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Default unblock handler

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_unblock_default( resmgr_context_t * ctp,
               io_pulse_t * msg,
               iofunc_ocb_t * ocb );
```

Arguments:

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg** A pointer to the `io_pulse_t` structure that describes the pulse that the resource manager received.
- **ocb** A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_unblock_default()` function calls `iofunc_unblock()`.

The `iofunc_unblock_default()` function implements the functionality required when the client requests to be unblocked (e.g. a signal or timeout).

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `unblock` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The unblock message is synthesized by the resource-manager shared library when a client wishes to unblock from its `MsgSendv()` to the
iofunc_unblock_default()

resource manager. The `iofunc_unblock_default()` function takes care of freeing up any locks that the client may have placed on the resource.

Returns:

- `RESMGR_DEFAULT`
  - No client connection was found.
- `RESMGR_NOREPLY`
  - A client connection has been unblocked.

Examples:

If you're calling `iofunc_lock_default()`, your unblock handler should call `iofunc_unblock_default()`:

```c
if((status = iofunc_unblock_default(...)) != _RESMGR_DEFAULT) {
    return status;
}
/* Do your own thing to look for a client to unblock */
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

iofunc_func_init(), iofunc_lock_default(), iofunc_ocb_t,
resmgr_attach(), resmgr_context_t, resmgr_io_funcs_t

Writing a Resource Manager chapter of the Programmer’s Guide.
### iofunc_unlink()

Verify that an entry can be unlinked

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_unlink( resmgr_context_t* ctp,
    io_unlink_t* msg,
    iofunc_attr_t* attr,
    iofunc_attr_t* dattr,
    struct _client_info* info );
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **msg** A pointer to the `io_unlink_t` structure that contains the message that the resource manager received; see below.

- **attr** A pointer to the `iofunc_attr_t` structure that describes the characteristics of the resource.

- **dattr** NULL, or a pointer to the `iofunc_attr_t` structure that describes the characteristics of the parent directory.

- **info** NULL, or a pointer to a `_client_info` structure that contains information about the client. For information about this structure, see `ConnectClientInfo()`.

**Library:**

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_unlink()` function verifies that the `msg` specifies valid semantics for an unlink, and that the client is allowed to unlink the resource, as specified by a combination of who the client is (`info`), and the resource attributes `attr`, `dattr`, `attr->uid` and `attr->gid`. 
If a directory entry is being removed, `iofunc_unlink()` checks to see that the directory is empty. The `iofunc_unlink()` function also updates the time stamps, and decrements the link count for the entry.

**io_unlink_t structure**

The `io_unlink_t` structure holds the `IO_CONNECT` message received by the resource manager:

```c
typedef union {
    struct _io_connect connect;
    struct _io_connect_link_reply link_reply;
    struct _io_connect_ftype_reply ftype_reply;
} io_unlink_t;
```

This message structure is a union of an input message (coming to the resource manager), `io_connect`, and two possible output or reply messages (going back to the client):

- **`io_connect_link_reply`** if the reply is redirecting the client to another resource
  
  Or:

- **`io_connect_ftype_reply`** if the reply consists of a status and a file type.

**Returns:**

- **EOK** Successful completion.
- **ENOTDIR** Attempt to unlink a nondirectory entry using directory semantics, (e.g. `rmdir file`).
- **EINVAL** Attempt to remove the "." directory.
- **ENOTEMPTY** Attempt to remove a directory that isn’t empty.
- **EROFS** Attempt to remove an entry on a read-only filesystem.
- **EACCES** The client doesn’t have permissions to do the operation.
- **EPERM** The group ID or owner ID didn’t match.
iofunc_unlink()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

_io_connect, _io_connect_link_reply, _io_connect_ftype_reply, ConnectClientInfo(), iofunc_attr_t, iofunc_check_access(), resmgr_context_t

Writing a Resource Manager chapter of the Programmer's Guide.
Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_unlock_ocb_default(
    resmgr_context_t * ctp,
    void * reserved,
    iofunc_ocb_t * ocb);
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `reserved` This argument must be NULL.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_unlock_ocb_default()` function calls `iofunc_attr_unlock()` to enforce unlocking on the attributes for the group of messages that were sent by the client.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `unlock_ocb` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.
Returns:

- **EOK**: Success.
- **EAGAIN**: On the first use, all kernel mutex objects were in use.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `iofunc_attr_unlock()`, `iofunc_func_init()`, `iofunc_ocb_t`, `resmgr_attach()`, `resmgr_context_t`, `resmgr_io_funcs_t`  

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

---

September 10, 2007
iofunc_utime()
Update time stamps

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_utime( resmgr_context_t* ctp,
                  io_utime_t* msg,
                  iofunc_ocb_t* ocb,
                  iofunc_attr_t* attr );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_utime_t` structure that contains the message that the resource manager received; see below.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `attr` A pointer to the `iofunc_attr_t` structure that describes the characteristics of the device that’s associated with your resource manager.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_utime()` helper function examines the `flags` member in the passed `attr` structure and sets the IOFUNC_ATTR_ATIME and IOFUNC_ATTR_MTIME bits if requested.

The function sets the IOFUNC_ATTR_CTIME and IOFUNC_ATTR_DIRTY_TIME bits. It then calls `iofunc_time_update()` to update the file times.
io_utime_t structure

The io_utime_t structure holds the _IO_UTIME message received by the resource manager:

```c
struct _io_utime {
    uint16_t type;
    uint16_t combine_len;
    int32_t cur_flag;
    struct utimbuf times;
};
```

typedef union {
    struct _io_utime i;
} io_utime_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, i.

The i member is a structure of type _io_utime that contains the following members:

- **type** _IO_UTIME.
- **combine_len** If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
- **cur_flag** If set, iofunc_utime() ignores the times member, and set the appropriate file times to the current time.
- **times** A utimbuf structure that specifies the time to use when setting the file times. For more information about this structure, see utime().

Returns:

- **EACCES** The client doesn’t have permissions to do the operation.
- **EFAULT** A fault occurred when the kernel tried to access the info buffer.
iofunc_utime()

EINVAL  The client process is no longer valid.
ENOSYS  NULL was passed in info.
EOK     Successful completion.
EPERM   The group ID or owner ID didn’t match.
EROFS   Attempt to remove an entry on a read-only filesystem.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
iofunc_time_update(), iofunc_utime_default(), utime()
Writing a Resource Manager chapter of the Programmer’s Guide.
Synopsis:

```
#include <sys/iofunc.h>

int iofunc_utime_default( resmgr_context_t* ctp,  
                          io_utime_t* msg,  
                          iofunc_ocb_t* ocb );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg` A pointer to the `io_utime_t` structure that contains the message that the resource manager received; see `iofunc_utime()`.
- `ocb` A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

Library:

```
libc
```

Use the `-l` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iofunc_utime_default()` function implements POSIX semantics for the client’s `utime()` call, which is received as an _IO_UTIME message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `utime` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_utime_default()` function calls `iofunc_utime()` to do the actual work. It verifies that the client has the necessary permissions to
effect a utime on the device. If so, the utime is performed, modifying elements of the \textit{ocb->attr} structure. This function takes care of updating these bits in the flags member of \texttt{ocb->attr}:

- \texttt{IOFUNC\_ATTR\_ATIME}
- \texttt{IOFUNC\_ATTR\_CTIME}
- \texttt{IOFUNC\_ATTR\_MTIME}
- \texttt{IOFUNC\_ATTR\_DIRTY\_TIME}
- \texttt{IOFUNC\_ATTR\_DIRTY\_MODE}

The \texttt{iofunc_utime()} function then calls \texttt{iofunc_time_update()} to update the appropriate time fields in \texttt{ocb->attr}.

**Returns:**

- \texttt{EOK} Successful completion.
- \texttt{EROFS} An attempt was made to utime a read-only filesystem.
- \texttt{EACCES} The client doesn’t have permissions to do the operation.
- \texttt{EPERM} The group ID or owner ID didn’t match.

**Classification:**

QNX Neutrino

**Safety**

\begin{tabular}{ll}
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}
See also:

`iofunc_func_init()`, `iofunc_time_update()`, `iofunc_ocb_t`,
`iofunc_utime()`, `resmgr_attach()`, `resmgr_context_t`,
`resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
**iofunc_write_default()**

 Default handler for _IO_WRITE messages

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_write_default( resmgr_context_t* ctp,
                        io_write_t* msg,
                        iofunc_ocb_t* ocb );
```

**Arguments:**

- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg** A pointer to the `io_write_t` structure that contains the message that the resource manager received. For more information, see `iofunc_write_verify()`.
- **ocb** A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.

**Library:**

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `iofunc_write_default()` function implements POSIX semantics for the client’s `write()` call, which is received as an _IO_WRITE message by the resource manager.

You can place this function directly into the `io_funcs` table passed to `resmgr_attach()`, at the `write` position, or you can call `iofunc_func_init()` to initialize all of the functions to their default values.

The `iofunc_write_default()` function calls `iofunc_write_verify()` to do the actual work.
Returns:

- EBADF: The client doesn’t have read access to this resource.
- EINVAL: An unknown xtype was given.
- EOK: The client has read access to this resource.

Classification:

QNX Neutrino

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

See also:

- iofunc_func_init(), iofunc_ocb_t, iofunc_open(), iofunc_write_verify(), resmgr_attach(), resmgr_context_t, resmgr_io_funcs_t

Writing a Resource Manager chapter of the Programmer’s Guide.
**iofunc_write_verify()**

Verify a client's write access to a resource

**Synopsis:**

```c
#include <sys/iofunc.h>

int iofunc_write_verify( resmgr_context_t* ctp,
                        io_write_t* msg,
                        iofunc_ocb_t* ocb,
                        int* nonblock );
```

**Arguments:**

- `ctp`: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- `msg`: A pointer to the `io_write_t` structure that contains the message that the resource manager received; see below.
- `ocb`: A pointer to the `iofunc_ocb_t` structure for the Open Control Block that was created when the client opened the resource.
- `nonblock`: NULL, or a pointer to a location where the function can store a value that indicates whether or not the device is nonblocking:
  - Nonzero — the client doesn’t want to be blocked (i.e. O_NONBLOCK was set).
  - Zero — the client wants to be blocked.

**Library:**

-libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
iofunc_write_verify()

Description:

The `iofunc_write_verify()` function checks that the client that sent the write message actually has write access to the resource, and, optionally (if `nonblock` isn’t NULL), sets `nonblock` to `O_NONBLOCK` or 0.

The write permission check is done against `ocb->ioflag`.

Note that the `io_write_t` message has an override flag called `msg->i.xtype`. This flag allows the client to override the default blocking behavior for the resource on a per-request basis. This override flag is checked, and returned in the optional `nonblock`.

In write callout for your resource manager, you’ll need to indicate how many bytes were written. You can do this with the macro:

```c
_IO_SET_WRITE_NBYTES(resmgr_context_t *ctp, int nbytes)
```

io_write_t structure

The `io_write_t` structure holds the _IO_WRITE message received by the resource manager:

```c
struct _io_write {
    uint16_t type;
    uint16_t combine_len;
    int32_t nbytes;
    uint32_t xtype;
    uint32_t zero;
    /* unsigned char data[nbytes]; */
};
```

typedef union {
    struct _io_write i;
    /* nbytes is returned with MsgReply */
} io_write_t;

The I/O message structures are unions of an input message (coming to the resource manager) and an output or reply message (going back to the client). In this case, there’s only an input message, `i`.

The `i` member is a structure of type `_io_write` that contains the following members:
iofunc_write_verify()

**type**

_IO_WRITE.

**combine_len**

If the message is a combine message, _IO_COMBINE_FLAG is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the *Programmer’s Guide*.

**nbytes**

The number of bytes that the client wants to write.

**xtype**

Extended type information; one of:

- _IO_XTYPE_NONE
- _IO_XTYPE_READCOND
- _IO_XTYPE_MQUEUE
- _IO_XTYPE_TCPIP
- _IO_XTYPE_TCPIP_MSG
- _IO_XTYPE_OFFSET
- _IO_XTYPE_REGISTRY
- _IO_XFLAG_DIR_EXTRA_HINT — this flag is valid only when reading from a directory. The filesystem should normally return extra directory information when it’s easy to get. If this flag is set, it is a hint to the filesystem to try harder (possibly causing media lookups) to return the extra information. The most common use would be to return _DTYPE_LSTAT information.
- _IO_XFLAG_NONBLOCK
- _IO_XFLAG_BLOCK

For more information, see “Handling other read/write details” in the Writing a Resource Manager chapter of the *Programmer’s Guide*.

The commented-out declaration for *data* indicates that *nbytes* bytes of data immediately follow the _io_write structure.
Returns:

EOK      The client has write access to this resource.
EBADF   The client doesn’t have write access to this resource.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`iofunc_read_verify()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Arm a resource manager

Synopsis:

```c
#include <unistd.h>
#include <sys/iomsg.h>

int ionotify ( int fd, 
              int action, 
              int flags, 
              const struct sigevent* event );
```

Arguments:

- `fd` The file descriptor associated with the resource manager that you want to notify.
- `action` The type of arming action to take; see “Actions,” below.
- `flags` The types of conditions that can be checked for notification; see “Flags,” below.
- `event` A pointer to a `sigevent` structure that defines the event that you want the resource manager to send as a notification, or NULL to disarm a notification.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ionotify()` function arms the resource manager associated with `fd` to send the event notification `event`. The event is sent when a condition specified by a combination of `action` and `flags` occurs.

Flags

The `flags` argument specifies the types of conditions that can be checked for notification. Each resource manager maintains a different context for each notification condition. Only those notification bits
specified are affected. In the following example, the second call to
ionotify() doesn’t affect the first, since it specifies a different
notification:

```c
ionotify( fd, _NOTIFY_ACTION_POLLARM,
        _NOTIFY_COND_INPUT, &event );
ionotify( fd, _NOTIFY_ACTION_POLLARM,
        _NOTIFY_COND_OUTPUT, &event );
```

The conditions specified by `flags` are:

- **_NOTIFY_COND_OBAND**
  Out-of-band data is available. The definition of out-of-band
data depends on the resource manager.

- **_NOTIFY_COND_OUTPUT**
  There’s room in the output buffer for more data. The amount of
  room available needed to satisfy this condition depends on the
  resource manager. Some resource managers may default to an
  empty output buffer, while others may choose some percentage
  of the buffer empty.

- **_NOTIFY_COND_INPUT**
  There’s input data available. The amount of data available
  defaults to 1. For a character device such as a serial port, this
  would be a character. For a POSIX message queue, it would be
  a message. Each resource manager selects an appropriate
  object.

The method for changing the default number for
__NOTIFY_COND_OUTPUT and __NOTIFY_COND_INPUT depends on
the device. For example, character special devices can call
`readcond()`.

For resource managers that support both an edited and raw mode, the
mode should be set to raw to ensure proper operation of `ionotify()`.

The above flags are located in the top bits of `flags`. They are defined
by __NOTIFY_COND_MASK.
In the case of an asynchronous notification using the passed `event`, such as a Neutrino pulse or queued realtime signal, the 32-bit value in `event->sigev_value.sival_int` is returned to you unmodified, unless you’ve selected the SI_NOTIFY code, in which case the top bits (defined by `_NOTIFY_COND_MASK`) are set to the active notifications. In this case, you should limit the `sival_int` to the mask defined by `_NOTIFY_DATA_MASK`.

For example, the Unix `select()` function specifies SI_NOTIFY and uses the allowable data bits of `sival_int` as a serial number.

If you’re using the SI_NOTIFY code, then you should clear the bits as specified by `_NOTIFY_COND_MASK` in the `sigev_value` field — the resource manager only ever ORs in a value, it never clears the bits.

**Actions**

The `action` argument specifies the type of arming action to take. When a condition is armed, the resource manager monitors it and, when met, delivers `event` using `MsgDeliverEvent()`. When an event is delivered, it’s always disarmed except where noted below.

Note that for transition arming (as specified by an `action` of `_NOTIFY_ACTION_TRANARM`, only one notification of that type can be outstanding per device. When the transition arm fires, it’s removed.

Each action is designed to support a specific notification type as follows:

- **_NOTIFY_ACTION_POLL**
  This action does a poll of the notification conditions specified by `flags`. It never arms an event, and it cancels all other asynchronous event notifications set up by a previous call to `ionotify()`. This also allows it to be used as a simple “disarm” call.
  Returns active conditions as requested by `flags`. 
_NOTIFY_ACTION_POLLARM

This action does a poll in the same way as _NOTIFY_ACTION_POLL. However, if none of the conditions specified in flags are present then each condition specified in flags is armed. If any condition is met, none of the conditions are armed. The Unix select() function uses ionotify() with this action.

Returns active conditions as requested by flags.

_NOTIFY_ACTION_TRANARM

This action arms for transitions of the notification conditions specified by flags. A transition is defined as a data transition from empty to nonempty on input. Its use on output isn’t defined. Note that if there is data available when this call is used, a data transition won’t occur. To generate an event using this type of notification, you must arm the event and then drain the input using a nonblocking read. After this point, new input data causes the event to be delivered. The mq_notify() function uses ionotify() with this action.

Since this arms for a transition, the return value is always zero.

You can use the _NOTIFY_ACTION_POLLARM or _NOTIFY_ACTION_POLL action to generate events that are level- as opposed to transition-oriented.

When an action is armed in a resource manager, it remains armed until:

- A thread sets a new action (this disarms any current action and possibly arms a new action),
- The event is delivered and the action wasn’t a continuous one,
- The thread closes the device.
Returns:

Active conditions as requested by flags. In the case of a transition action, a zero is returned. If an error occurs, -1 is returned (errno is set).

Errors:

- EBADF: The connection indicated by fd doesn’t exist, or fd is no longer connected to a channel.
- EFAULT: A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.
- EINVAL: The call was interrupted by a signal.
- ENOMEM: The resource manager couldn’t allocate a notify entry to save the request.
- ENOSYS: The requested action isn’t supported by this resource manager.
- ESRVFault: A fault occurred in a server’s address space while accessing the server’s message buffers. This may have occurred on the receive or the reply.
- ETIMEDOUT: A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
## ionotify()

### Safety

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

### See also:

- **sigevent**
Internet Protocol

Synopsis:

#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_RAW,
            proto );

Description:

IP is the transport layer protocol used by the Internet protocol family. You may set options at the IP level when you're using higher-level protocols based on IP, such as TCP and UDP. You may also access IP through a “raw socket” (when you’re developing new protocols or special-purpose applications).

There are several IP-level setsockopt() and getsockopt() options. You can use IP_OPTIONS to provide IP options to be transmitted in the IP header of each outgoing packet or to examine the header options on incoming packets. IP options may be used with any socket type in the Internet family. The format of IP options to be sent is that specified by the IP protocol specification (RFC-791), with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address is extracted from the option list and the size adjusted accordingly before use. To disable previously specified options, use a zero-length buffer:

setsockopt(s, IPPROTO_IP, IP_OPTIONS, NULL, 0);

You can use IP_TOS and IP_TTL to set the type-of-service and time-to-live fields in the IP header for SOCK_STREAM and SOCK_DGRAM sockets. For example:

int tos = IPTOS_LOWDELAY; /* see <netinet/ip.h> */
setsockopt(s, IPPROTO_IP, IP_TOS, &tos, sizeof(tos));

int ttl = 60; /* max = 255 */
setsockopt(s, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
If the IP_RECVSTADDR option is enabled on a SOCK_DGRAM or SOCK_RAW socket, the recvmsg() call returns the destination IP address for a UDP datagram. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by the IP address. The cmsghdr fields have the following values:

\[\text{cmsg_len} = \text{sizeof(struct cmsghdr) + sizeof(struct in_addr)}\]
\[\text{cmsg_level} = \text{IPPROTO_IP}\]
\[\text{cmsg_type} = \text{IP_RECVDSTADDR}\]

If the IP_RECVIF option is enabled on a SOCK_DGRAM or SOCK_RAW socket, the recvmsg() call returns a struct sockaddr_dl corresponding to the interface on which the packet was received. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by the struct sockaddr_dl. The cmsghdr fields have the following values:

\[\text{cmsg_len} = \text{sizeof(struct cmsghdr) + sizeof(struct sockaddr_dl)}\]
\[\text{cmsg_level} = \text{IPPROTO_IP}\]
\[\text{cmsg_type} = \text{IP_RECVIF}\]

Raw IP sockets are connectionless, and are normally used with the sendto() and recvfrom() calls, although you can also use connect() to fix the destination for future packets (in which case you can use the read() or recv() and write() or send() system calls).

If the proto parameter to socket() is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If proto is nonzero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with), unless the IP_HDRINCL option has been set. Incoming packets are received with IP header and options intact.

IP_HDRINCL indicates the complete IP header is included with the data and may be used only with the SOCK_RAW type.

```
#include <netinet/ip.h>

int hincl = 1; /* 1 = on, 0 = off */
setsockopt(s, IPPROTO_IP, IP_HDRINCL, &hincl, sizeof(hincl));
```
The program must set all the fields of the IP header, including the following:

```c
ip->ip_v = IPVERSION;
ip->ip_hl = hlen >> 2;
ip->ip_id = 0; /* 0 means kernel set appropriate value */
ip->ip_off = offset;
```

If the header source address is set to INADDR_ANY, the kernel chooses an appropriate address.

**Multicasting**

IP multicasting is supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on networks where the interface driver supports multicasting.

**Multicast Options**

**IP_MULTICAST_TTL**

Change the time-to-live (TTL) for outgoing multicast datagrams in order to control the scope of the multicasts:

```c
u_char ttl; /* range: 0 to 255, default = 1 */
setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Datagrams with a TTL of 1 aren’t forwarded beyond the local network. Multicast datagrams with a TTL of 0 aren’t transmitted on any network, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback hasn’t been disabled on the sending socket (see below). Multicast datagrams with TTL greater than 1 may be forwarded to other networks if a multicast router is attached to the local network.

**IP_MULTICAST_IF**

For hosts with multiple interfaces, each multicast transmission is sent from the primary network interface. The IP_MULTICAST_IF option overrides the default for subsequent transmissions from a given socket:

```c
struct in_addr addr;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```
where \textit{addr} is the local IP address of the desired interface or \texttt{INADDR\_ANY} to specify the default interface. You can get an interface’s local IP address and multicast capability by sending the SIOCGIFCONF and SIOCGIFFLAGS requests to \texttt{ioctl()}. Normal applications shouldn’t need to use this option.

**IP\_MULTICAST\_LOOP**

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. The IP\_MULTICAST\_LOOP option gives the sender explicit control over whether or not subsequent datagrams are looped back:

\begin{verbatim}
  u_char loop; /* 0 = disable, 1 = enable (default) */
  setsockopt(s, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
\end{verbatim}

This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions. It shouldn’t generally be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender doesn’t belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

**IP\_ADD\_MEMBERSHIP**

A host must become a member of a multicast group before it can receive datagrams sent to the group. To join a multicast group, use the IP\_ADD\_MEMBERSHIP option:

\begin{verbatim}
  struct ip_mreq mreq;
  setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
\end{verbatim}
where \( mreq \) is the following structure:

```c
struct ip_mreq {
    struct in_addr imr_multiaddr; /* multicast group to join */
    struct in_addr imr_interface; /* interface to join on */
}
```

Set \( imr \_interface \) to INADDR\_ANY to choose the default multicast interface, or to the IP address of a particular multicast-capable interface if the host is multihomed. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface. Up to IP\_MAX\_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

**IP\_DROP\_MEMBERSHIP**

To drop a membership, use:

```c
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP\_DROP\_MEMBERSHIP, &mreq, sizeof(mreq));
```

where \( mreq \) contains the same values as used to add the membership. Memberships are dropped when the socket is closed or the process exits.

**Returns:**

A descriptor referencing the socket, or -1 if an error occurs (\( errno \) is set).

**Errors:**

- **EADDRNOTAVAIL**
  
  You tried to create a socket with a network address for which no network interface exists.

- **EISCONN**
  
  You tried to establish a connection on a socket that already has one or to send a datagram with the destination address specified, but the socket is already connected.
ENOBUFS The system ran out of memory for an internal data structure.

ENOTCONN You tried to send a datagram, but no destination address was specified and the socket hasn’t been connected.

The following error specific to IP may occur when setting or getting IP options:

EINVAL An unknown socket option name was given. The IP option field was improperly formed — an option field was shorter than the minimum value or longer than the option buffer provided.

See also:

ICMP protocol

connect(), getsockopt(), ioctl(), read(), recv(), recvfrom(), recvmsg(), send(), sendto(), setsockopt(), socket(), write()

RFC 791
IPsec
Internet security protocol

Synopsis:

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <netinet6/ipsec.h>

int socket( PF_KEY,
            SOCK_RAW,
            PF_KEY_V2 );
```

Description:

IPsec is a security protocol for the Internet Protocol layer. It consists of these sub-protocols:

AH (Authentication Header)

Guarantees the integrity of the IP packet and protects it from intermediate alteration or impersonation by attaching a cryptographic checksum computed by one-way hash functions.

ESP (Encapsulated Security Payload)

Protects the IP payload from wire-tapping by encrypting it using secret-key cryptography algorithms.

IPsec has these modes of operation:

- Transport — protects peer-to-peer communication between end nodes.
- Tunnel — supports IP-in-IP encapsulation operation and is designed for security gateways, like VPN configurations.

Kernel interface

The IPsec protocol behavior is controlled by these engines:

- Key management engine — accessed from an application using PF_KEY sockets. The RFC 2367 specification defines the PF_KEY socket API.
• Policy engine — accessed with the PF_KEY API, the `setsockopt()` operations, and the `sysctl()` interface. (The `sysctl` utility is a cover for the `sysctl()` function.) The `setsockopt()` function defines per-socket behavior and the `sysctl()` interface defines host-wide default behavior.

These engines are located in the socket manager. The socket manager implements the PF_KEY interface and allows you to define IPsec policy similar to per-packet filters. Note that the socket manager code doesn’t implement the dynamic encryption key exchange protocol IKE (Internet Key Exchange) — that implementation should be done at the application level (usually as daemons), using the previously described APIs.

**Policy management**

The socket manager implements experimental policy management. You can manage the IPsec policy in these ways:

• Configure a per-socket policy using `setsockopt()`.

• Configure the socket manager packet filter-based policy using the PF_KEY interface or via the `setkey` utility.

  In this case, the default policy is allowed with the `setkey`. By configuring the policy to `default`, you can use the system-wide `sysctl` utility variables. (The `sysctl` utility displays various runtime options.)

  If the socket manager finds no matching policy, the system-wide default value is applied.

  For a list of `net.inet6.ipsec6.*` variables, see the `sysctl` utility in the Utilities Reference.

**Miscellaneous `sysctl` variables**

The following variables are accessible via the `sysctl` utility for tweaking socket manager IPsec behavior:
The variables are interpreted as follows:

\textit{ipsec.ah\_cleartos}

When computing \textit{AH} authentication data, the socket manager clears the type-of-service field in the IPv4 header if the value is set to a nonzero value. The variable tweaks \textit{AH} behavior to interoperate with devices that implement \textit{RFC 1826 AH}. Set this to a nonzero value (clear the type-of-service field) if you want to conform to \textit{RFC 2402}.

\textit{ipsec.ah\_offsetmask}

When computing \textit{AH} authentication data, the socket manager includes the 16-bit fragment offset field (including flag bits) in the IPv4 header, after computing a logical “AND” with the variable. This variable tweaks the \textit{AH} behavior to interoperate with devices that implement \textit{RFC 1826 AH}. Set this value to zero (clear the fragment offset field during computation) if you want to conform to \textit{RFC 2402}.

\textit{ipsec.dfbits}

Configures the socket manager behavior for IPv4 IPsec tunnel encapsulation. The variable is supplied to conform to \textit{RFC 2403} Chapter 6.1.
If the value is set to:  Then:

0  The DF bit on the outer IPv4 header is cleared.
1  The outer DF bit on the header is set from the inner DF bit.
2  The DF bit is copied from the inner header to the outer.

`ipsec.ecn`  If set to nonzero, the IPv4 IPsec tunnel encapsulation/decapsulation behavior supports ECN (Explicit Congestion Notification), as documented in the IETF draft `draft-ietf-ipsec-ecn-02.txt`.

`ipsec.debug`  If set to nonzero, debug messages are generated to the `syslog`.

Variables under the `net.inet6.ipsec6` tree have meaning similar to their `net.inet.ipsec` counterparts.

Protocols

Because the IPsec protocol works like a plugin to the INET and INET6 protocols, IPsec supports most of the protocols defined upon those IP-layer protocols. Some of the protocols, like ICMP or ICMP6, may behave differently with IPsec. This is because IPsec can prevent ICMP or ICMP6 routines from looking into the IP payload.

Setting the policy

You can set the policy manually by calling `setkey`, or set it permanently in `/etc/inetd.conf`. Valid policy settings include:

```
for setkey:  -P direction discard
             -P direction ipsec request ...
             -P direction none
```
for /etc/inetd.conf:

- **direction** bypass
- **direction** entrust
- **direction** ipsec request ...

where:

- **direction**: The direction in which the policy is applied. It’s either `in` or `out`.
- **bypass** (/etc/inetd.conf only) Bypass the IPsec processing and transmit the packet in clear text. This option is for privileged sockets.
- **discard** (setkey only) Discard the packet matching indexes.
- **entrust** (/etc/inetd.conf only) Consult the Security Policy Database (SPD) in the stack. The SPD is set by `setkey` (see the Utilities Reference).
- **ipsec request** ...

Put the IPsec operation into the packet. You can specify one or more `request` strings using the following format:

```
protocol/mode/src-dst [/level]
```

For detailed descriptions of the arguments in the `request` string, see below.

- **none** (setkey only) Don’t put the IPsec operation into the packet.

**Arguments for request**

- **protocol**: One of:
  - `ah` — Authentication Header. Guarantees the integrity of the IP packets and protects them from intermediate alteration or impersonation, by attaching cryptographic checksums computed by one-way hash functions.
- **esp** — Encapsulated Security Payload. Protects the IP payload from wire-tapping by encrypting it with secret key cryptography algorithms.
- **ipcomp** — IP Payload Compression Protocol.

**mode**

Security protocol to be used, which is one of:

- **transport** — Protects peer-to-peer communication between end nodes.
- **tunnel** — Includes IP-in-IP encapsulation operations and is designed for security gateways, like VPN configurations.

**dst**, **src**

The “receiving node” (**dst**) and “sending node” (**src**) endpoint addresses of the Security Association (SA). When the direction specified is **in**, **dst** would represent this node and **src** the other node (peer).

If **transport** is specified as the **mode**, you can omit these values.

**level**

One of:

- **default** — The stack should consult the system default policy that’s set by the **sysctl** utility.
- **require** — An SA is required whenever the kernel deals with the packet.
- **use** — Use an SA if it’s available; otherwise, keep the normal operation.
- **unique** — (**setkey** only) Similar to **require**, but adds the restriction that the SA for outbound traffic is used only for this policy.

You may need the identifier in order to relate the policy and the SA when you define the SA by manual keying. You can put the decimal number as the identifier after **unique**, such as:

```
unique: number
```
The value of `number` must be between 1 and 32767. If the request string is kept unambiguous, the `level` and slash prior to `level` can be omitted. However, you should specify them explicitly to avoid unintended behaviors.

If the `level` isn’t specified in the `setkey` command, `unique` is used by default.

**Caveats:**

The IPsec support is subject to change as the IPsec protocols develop. There’s no single standard for policy engine API, so the policy engine API described herein is just for KAME implementation.

The AH tunnel may not work as you might expect. If you configure the `require` policy against AH tunnel for inbound, tunneled packets will be rejected. This is because AH authenticates the encapsulating (outer) packet, not the encapsulated (inner) packet.

Under certain conditions, a truncated result may be returned from the socket manager from SADB_DUMP and SADB_SPDDUMP operations on a PF_KEY socket. This occurs if there are too many database entries in the socket manager and the socket buffer for the PF_KEY socket is too small. If you manipulate many IPsec key/policy database entries, increase the size of socket buffer.

**See also:**

ICMP, ICMP6, INET6, IP, IPv6 protocols

`ioctl()`, `socket()`, `sysctl()`

`/etc/inetd.conf`, `setkey` in the *Utilities Reference*.

*RFC 2367, RFC 1826, RFC 2402, RFC 2403*

Detailed documentation about the IP security protocol may be found at the IPsec FAQ website at [http://www.netbsd.org/Documentation/network/ipsec/](http://www.netbsd.org/Documentation/network/ipsec/).
Synopsis:

```c
#include <netinet6/ipsec.h>

char* ipsec_dump_policy(char *buf,
                        char *delim);
```

Arguments:

- `buf`: A pointer to an IPsec policy structure `struct sadb_x_policy`.
- `delim`: Delimiter string, usually a NULL which indicates a space (" ").

Library:

- `libipsec`

Use the `-l ipsec` option to `qcc` to link against this library.

Description:

The function `ipsec_dump_policy()` generates a readable string from an IPSEC policy specification. Please refer to `ipsec_set_policy()` for details about the policies.

The `ipsec_dump_policy()` function converts IPsec policy structure into a readable form. Therefore, `ipsec_dump_policy()` is the inverse of `ipsec_set_policy()`. If you set `delim` to NULL, a single whitespace is assumed. The function `ipsec_dump_policy()` returns a pointer to a dynamically allocated string. It is the caller’s responsibility to reclaim the region, by using `free()`.

Returns:

A pointer to dynamically allocated string, or NULL if an error occurs.
ipsec_dump_policy()

Examples:

See ipsec_set_policy().

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

IPsec, ipsec_get_policylen(), ipsec_set_policy(), ipsec_strerror()

setkey in the Utilities Reference
**Synopsis:**

```c
#include <netinet6/ipsec.h>

int ipsec_get_policylen(char *buf);
```

**Arguments:**

- `buf` A pointer to an IPsec policy structure `struct sadb_x_policy`.

**Library:**

`libipsec`

Use the `-l ipsec` option to `qcc` to link against this library.

**Description:**

The function `ipsec_get_policylen()` gets the length of the IPsec policy. Please refer to `ipsec_set_policy()` for details about the policies.

You may want the length of the generated buffer when calling `setsockopt()`. The function `ipsec_get_policylen()` returns the length.

**Returns:**

The size of the buffer, or a negative value if an error occurs.

**Examples:**

See `ipsec_set_policy()`.

**Classification:**

Unix
ipsec_get_policylen()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

IPsec, ipsec_dump_policy(), ipsec_set_policy(), ipsec_strerror()

`setkey` in the Utilities Reference
ipsec_set_policy()

Generate an IPsec policy specification structure from a readable string

Synopsis:

```c
#include <netinet6/ipsec.h>

char* ipsec_set_policy(char *policy,
                        int len);
```

Arguments:

- `len`: The length of the policy string.
- `policy`: A string that describes a `struct sadb_x_policy` and optionally a `struct sadb_x_ipsecrequest`, formatted as described below.

Library:

- `libipsec`

Use the `-l ipsec` option to `qcc` to link against this library.

Description:

The function `ipsec_set_policy()` generates an IPsec policy specification structure, namely a `struct sadb_x_policy` and potentially a `struct sadb_x_ipsecrequest` from a human-readable policy specification. This function returns a pointer to the IPsec policy specification structure.

You should release the buffer returned by `ipsec_set_policy()` by calling `free()`. See the example below.

The policy is formatted as one of the following:

- **direction discard**
  - The direction must be `in` or `out`. It specifies which direction the policy needs to be applied. With the discard policy, packets are dropped if they match the policy.
direction entrust
   Consultation to SPD — defined by setkey.

direction bypass
   Bypass the IPsec processing, i.e. packets are transmitted in clear. This is for privileged sockets.

direction ipsec request ...
   The matching packets are subject to IPsec processing. The ipsec string can be followed by one or more request strings, which are formatted as below:

   protocol / mode / src - dst [/level]

protocol Either ah, esp, or ipcomp.
mode Either transport or tunnel.
src and dst The IPsec endpoints; src is the sending node and dst is the receiving node. Therefore, when direction is in, dst is this node and src is the other node (peer).
level Either default, use, require or unique.
   • default — the kernel should consult the system default policy defined by sysctl().
   • use — a relevant SA (security association) is used when available, since the kernel may perform IPsec operation against packets when possible. In this case, packets are transmitted in clear (when SA is not available), or encrypted (when SA is available).
   • require — a relevant SA is required, since the kernel must perform IPsec operation against packets.
   • unique is the same as require. However, it adds the restriction that the SA for outbound traffic is used only for this policy. You may need the identifier in order to relate the policy and the SA when you define the SA by manual
keying. You put the decimal number as the identifier like:

```
unique: number
```

where `number` must be between 1 and 32767. If the request string is kept unambiguous, you can omit the `level` and the slash (“/”) prior to `level`. However, you should specify them explicitly to avoid unintended behavior. If `level` is omitted, it will be interpreted as default.

Here's an example of policy information:

```
in discard
out ipsec esp/transport//require
in ipsec ah/transport//require
out ipsec esp/tunnel/10.1.1.2-10.1.1.1/use
in ipsec ipcom/transport//use esp/transport//use
```

It differs from the specification of `setkey`, where both `entrust` and `bypass` are not used. Please refer to `setkey` for detail.

**Returns:**

A pointer to the allocated policy specification, or NULL if an error occurs.

**Examples:**

```c
#include <netinet6/ipsec.h>
#include <sys/socket.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>

int main(void)
{
    char *sadb;
    char *policy = "in discard";
    int len;

    sadb = ipsec_set_policy(policy, strlen(policy));
}```
if (sadb == NULL) {
    fprintf(stderr, "ipsec_set_policy: \n", ipsec_strerror());
    return 1;
}

len = ipsec_get_policylen(sadb);
printf("len: %d\n", len);

policy = NULL;
policy = ipsec_dump_policy(sadb, NULL);

if (policy == NULL) {
    fprintf(stderr, "ipsec_dump_policy: \n", ipsec_strerror());
    return 1;
}

printf("policy: \n", policy);
free(policy);
free(sadb);
return 0;
}

Classification:

Unix

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

IPsec, ipsec_dump_policy(), ipsec_get_policylen(), ipsec_strerror()

setkey in the Utilities Reference
**Synopsis:**

```c
#include <netinet6/ipsec.h>

const char *
ipsec_strerror(void);
```

**Library:**

`libipsec`

Use the `-l ipsec` option to `qcc` to link against this library.

**Description:**

This `ipsec_strerror()` function is used to obtain the error message string from the last failed IPsec call.

**Returns:**

A pointer to an error message.

Don’t modify the string that this function returns.

**Examples:**

```c
#include <netinet6/ipsec.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *sadb;
    char *policy = "in discard";
    int len;

    sadb = ipsec_set_policy(policy, strlen(policy));

    if (sadb == NULL) {
```
ipsec_strerror()

```c
fprintf(stderr, "ipsec_set_policy: %s\n", ipsec_strerror());
return 1;
}
len = ipsec_get_policylen(sadb);
printf("len: %d\n", len);

policy = NULL;
policy = ipsec_dump_policy(sadb, NULL);
if (policy == NULL) {
    fprintf(stderr, "ipsec_dump_policy: %s\n", ipsec_strerror());
    return 1;
}
printf("policy: %s\n", policy);
free(policy);
free(sadb);
return 0;
}
```

**Classification:**

Unix

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `IPsec, ipsec_dump_policy(), ipsec_get_policylen(), ipsec_set_policy()`
- `setkey` in the Utilities Reference
Copyright © 2007, QNX Software Systems GmbH & Co. KG. IP6

Internet Protocol version 6

Synopsis:
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET6,
        SOCK_RAW,
        proto );

Description:
The IP6 protocol is the network-layer protocol used by the Internet Protocol version 6 family (AF_INET6). Options may be set at the IP6 level when using higher-level protocols based on IP6 (such as TCP and UDP). It may also be accessed through a “raw socket” when developing new protocols, or special-purpose applications.

There are several IP6-level setsockopt()/getsockopt() options. They are separated into the basic IP6 sockets API (defined in RFC2553), and the advanced API (defined in RFC2292). The basic API looks very similar to the API presented in IP. The advanced API uses ancillary data and can handle more complex cases.

Specifying some of the socket options requires root privileges.

Basic IP6 sockets API

You can use the IPV6_UNICAST_HOPS option to set the hoplimit field in the IP6 header on unicast packets. If you specify -1, the socket manager uses the default value. If you specify a value of 0 to 255, the packet uses the specified value as it hoplimit. Other values are considered invalid and result in an error code of EINVAL. For example:

int hlim = 60; /* max = 255 */
setsockopt( s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,
        &hlim, sizeof(hlim) );

The IP6 multicasting is supported only on AF_INET6 sockets of type SOCK_DGRAM and SOCK_RAW, and only on networks where the interface driver supports multicasting.
The IPV6_MULTICAST_HOPS option changes the hoplimit for outgoing multicast datagrams in order to control the scope of the multicasts:

```c
unsigned int hlim; /* range: 0 to 255, default = 1 */
setsockopt(s, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
           &hlim, sizeof(hlim));
```

Datagrams with a hoplimit of 1 aren’t forwarded beyond the local network. Multicast datagrams with a hoplimit of 0 won’t be transmitted on any network, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback hasn’t been disabled on the sending socket (see below). Multicast datagrams with a hoplimit greater than 1 may be forwarded to other networks if a multicast router is attached to the local network.

For hosts with multiple interfaces, each multicast transmission is sent from the primary network interface. The IPV6_MULTICAST_IF option overrides the default for subsequent transmissions from a given socket:

```c
unsigned int outif;
outif = if_nametoindex("ne0");
setsockopt(s, IPPROTO_IPV6, IPV6_MULTICAST_IF,
           &outif, sizeof(outif));
```

(The `outif` argument is an interface index of the desired interface, or 0 to specify the default interface.)

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP6 layer for local delivery. The IPV6_MULTICAST_LOOP option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```c
u_char loop; /* 0 = disable, 1 = enable (default) */
setsockopt(s, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
           &loop, sizeof(loop));
```

This option improves performance for applications that may have no more than one instance on a single host (such as a router daemon), by eliminating the overhead of receiving their own transmissions. Don’t use the IPV6_MULTICAST_LOOP option if there might be more than
one instance of your application on a single host (e.g. a conferencing program), or if the sender doesn’t belong to the destination group (e.g. a time-querying program).

A multicast datagram sent with an initial hoplimit greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such a delivery.

A host must become a member of a multicast group before it can receive datagrams sent to the group. To join a multicast group, use the IPV6_JOIN_GROUP option:

```c
struct ipv6_mreq mreq6;
setsockopt( s, IPPROTO_IPV6, IPV6_JOIN_GROUP,
            &mreq6, sizeof(mreq6) );
```

Note that the `mreq6` argument has the following structure:

```c
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;
    unsigned int ipv6mr_interface;
};
```

Set the `ipv6mr_interface` member to 0 to choose the default multicast interface, or set it to the interface index of a particular multicast-capable interface if the host is multihomed. Membership is associated with a single interface; programs running on multihomed hosts may need to join the same group on more than one interface.

To drop a membership, use:

```c
struct ipv6_mreq mreq6;
setsockopt( s, IPPROTO_IPV6, IPV6_LEAVE_GROUP,
            &mreq6, sizeof(mreq6) );
```

The `mreq6` argument contains the same values as used to add the membership. Memberships are dropped when the socket is closed or the process exits.

The IPV6_PORTRANGE option controls how ephemeral ports are allocated for SOCK_STREAM and SOCK_DGRAM sockets. For example:
int range = IPV6_PORTRANGE_LOW; /* see <netinet/in.h> */
setsockopt( s, IPPROTO_IPV6, IPV6_PORTRANGE, &range,
        sizeof(range) );

The IPV6_BINDV6ONLY option controls the behavior of the
AF_INET6 wildcard listening socket. The following example sets the
option to 1:

int on = 1;
setsockopt( s, IPPROTO_IPV6, IPV6_BINDV6ONLY,
        &on, sizeof(on) );

If you set the IPV6_BINDV6ONLY option to 1, the AF_INET6
wildcard listening socket accepts IP6 traffic only. If set to 0, the
socket accepts IPv4 traffic as well, as if it were from an IPv4 mapped
address, such as ::ffff:10.1.1.1. Note that if you set the option
to 0, IPv4 access control gets much more complicated. For example,
even if you have no listening AF_INET socket on port X, you’ll end up
accepting IPv4 traffic by an AF_INET6 listening socket on the same
port. The default value for this flag is copied at socket-instantiation
time, from the net.inet6.ip6.bindv6only variable from the sysctl
utility. The option affects TCP and UDP sockets only.

Advanced IP6 sockets API

The advanced IP6 sockets API lets applications specify or obtain
details about the IP6 header and extension headers on packets. The
advanced API uses ancillary data for passing data to or from the
socket manager.

There are also setsockopt() / getsockopt() options to get optional
information on incoming packets:

- IPV6_PKTINFO
- IPV6_HOPLIMIT
- IPV6_HOPOPTS
- IPV6_DSTOPTS
- IPV6_RTHDR
int on = 1;

setsockopt(fd, IPPROTO_IPV6, IPV6_PKTINFO, &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_HOPLIMIT, &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_HOPOPTS, &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_DSTOPTS, &on, sizeof(on));
setsockopt(fd, IPPROTO_IPV6, IPV6_RTHDR, &on, sizeof(on));

When any of these options are enabled, the corresponding data is returned as control information by `recvmsg()`, as one or more ancillary data objects.

If IPV6_PKTINFO is enabled, the destination IP6 address and the arriving interface index are available via `struct in6_pktinfo` on an ancillary data stream. You can pick the structure by checking for an ancillary data item by setting the `cmsg_level` argument to IPPROTO_IPV6 and the `cmsg_type` argument to IPV6_PKTINFO.

If IPV6_HOPLIMIT is enabled, the hoplimit value on the packet is made available to the application. The ancillary data stream contains an integer data item with a `cmsg_level` of IPPROTO_IPV6 and a `cmsg_type` of IPV6_HOPLIMIT.

The `inet6_option_space()` family of functions help you parse ancillary data items for IPV6_HOPOPTS and IPV6_DSTOPTS. Similarly, the `inet6_rthdr_space()` family of functions help you parse ancillary data items for IPV6_RTHDR.

The IPV6_HOPOPTS and IPV6_DSTOPTS values may appear multiple times on an ancillary data stream (note that the behavior is slightly different from the specification). Other ancillary data items appear no more than once.

You can pass ancillary data items with normal payload data, using the `sendmsg()` function. Ancillary data items are parsed by the socket manager, and are used to construct the IP6 header and extension.
headers. For the *cmsg_level* values listed above, the ancillary data format is the same as the inbound case.

Additionally, you can specify a IPV6_NEXTHOP data object. The IPV6_NEXTHOP ancillary data object specifies the next hop for the datagram as a socket address structure. In the *msgHdr* structure containing this ancillary data, the *cmsg_level* argument is *IPPROTO_IPV6*, the *cmsg_type* argument is IPV6_NEXTHOP, and the first byte of *cmsg_data* is the first byte of the socket address structure.

If the socket address structure contains an IP6 address (e.g. the *sin6_family* argument is AF_INET6), then the node identified by that address must be a neighbor of the sending host. If that address equals the destination IP6 address of the datagram, then this is equivalent to the existing SO_DONTROUTE socket option.

For applications that don’t, or can’t use the *sendmsg()* or the *recvmsg()* function, the IPV6_PKTOPTIONS socket option is defined. Setting the socket option specifies any of the optional output fields:

```c
setsockopt(fd, IPPROTO_IPV6, IPV6_PKTOPTIONS, &buf, len);
```

The *buf* argument points to a buffer containing one or more ancillary data objects; the *len* argument is the total length of all these objects. The application fills in this buffer exactly as if the buffer were being passed to the *sendmsg()* function as control information.

The options set by calling *setsockopt()* for IPV6_PKTOPTIONS are called “sticky” options because once set, they apply to all packets sent on that socket. The application can call *setsockopt()* again to change all the sticky options, or it can call *setsockopt()* with a length of 0 to remove all the sticky options for the socket.

The corresponding receive option:

```c
getsockopt(fd, IPPROTO_IPV6, IPV6_PKTOPTIONS, &buf, &len);
```

returns a buffer with one or more ancillary data objects for all the optional receive information that the application has previously specified that it wants to receive. The *buf* argument points to the buffer that the call fills in. The *len* argument is a pointer to a
value-result integer; when the function is called, the integer specifies the size of the buffer pointed to by buf, and on return this integer contains the actual number of bytes that were stored in the buffer. The application processes this buffer exactly as if it were returned by recvmsg() as control information.

**Advanced API and TCP sockets**

When using getsockopt() with the IPV6_PKTOPTIONS option and a TCP socket, only the options from the most recently received segment are retained and returned to the caller, and only after the socket option has been set. The application isn’t allowed to specify ancillary data in a call to sendmsg() on a TCP socket, and none of the ancillary data described above is ever returned as control information by recvmsg() on a TCP socket.

**Conflict resolution**

In some cases, there are multiple APIs defined for manipulating an IP6 header field. A good example is the outgoing interface for multicast datagrams: it can be manipulated by IPV6_MULTICAST_IF in the basic API, by IPV6_PKTINFO in the advanced API, and by the sin6_scope_id field of the socket address structure passed to the sendto() function.

In QNX Neutrino, when conflicting options are given to the socket manager, the socket manager gets the value in the following order:

1. options specified by using ancillary data
2. options specified by a sticky option of the advanced API
3. options specified by using the basic API
4. options specified by a socket address.

The conflict resolution is undefined in the API specification and depends on the implementation.
Raw IP6 Sockets

Raw IP6 sockets are connectionless, and are normally used with sendto() and recvfrom(), although you can also use connect() to fix the destination for future packets (in which case you can use read() or recv(), and write() or send()).

If proto is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If proto is nonzero, that protocol number is used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP6 header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received without the IP6 header or extension headers.

All data sent via raw sockets must be in network byte order; all data received via raw sockets is in network-byte order. This differs from the IPv4 raw sockets, which didn’t specify a byte ordering and typically used the host’s byte order.

Another difference from IPv4 raw sockets is that complete packets (i.e. IP6 packets with extension headers) can’t be read or written using the IP6 raw sockets API. Instead, ancillary data objects are used to transfer the extension headers, as described above.

All fields in the IP6 header that an application might want to change (i.e. everything other than the version number) can be modified using ancillary data and/or socket options by the application for output. All fields in a received IP6 header (other than the version number and Next Header fields) and all extension headers are also made available to the application as ancillary data on input. Hence, there’s no need for a socket option similar to the IPv4 IP_HDRINCL socket option.

When writing to a raw socket, the socket manager automatically fragments the packet if the size exceeds the path MTU, inserting the required fragmentation headers. On input, the socket manager reassembles received fragments, so the reader of a raw socket never sees any fragment headers.
Most IPv4 implementations give special treatment to a raw socket created with a third argument to `socket()` of `IPPROTO_RAW`, whose value is normally 255. We note that this value has no special meaning to an IP6 raw socket (and the IANA currently reserves the value of 255 when used as a next-header field).

For ICMP6 raw sockets, the socket manager calculates and inserts the mandatory ICMP6 checksum.

For other raw IP6 sockets (i.e. for raw IP6 sockets created with a third argument other than `IPPROTO_ICMPV6`), the application must:

1. Set the new `IPV6_CHECKSUM` socket option to have the socket manager compute and store a pseudo header checksum for output.
2. Verify the received pseudo header checksum on input, discarding the packet if the checksum is in error.

This option prevents applications from having to perform source-address selection on the packets they send. The checksum incorporates the IP6 pseudo-header, defined in Section 8.1 of RFC 2460. This new socket option also specifies an integer offset into the user data of where the checksum is located.

```c
int offset = 2;
setsockopt(fd, IPPROTO_IPV6, IPV6_CHECKSUM,
            &offset, sizeof(offset));
```

By default, this socket option is disabled. Setting the offset to -1 also disables the option. Disabled means:

1. The socket manager won’t calculate and store a checksum for outgoing packets.
2. The socket manager kernel won’t verify a checksum for received packets.
Since the checksum is always calculated by the socket manager for
an ICMP6 socket, applications can’t generate ICMPv6 packets
with incorrect checksums (presumably for testing purposes) using
this API.

- The IPV6_NEXTHOP object/option isn’t fully implemented.

See also:

`getsockopt()`, ICMP6 protocol, INET6 protocol, `recv()`, `send()`,
`setsockopt()`
isalnum()
Test a character to see if it’s alphanumeric

Synopsis:

```c
#include <ctype.h>

int isalnum( int c );
```

Arguments:

- `c` The character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `isalnum()` function tests if the argument `c` is an alphanumeric character (a to z, A to Z, or 0 to 9). An alphanumeric character is any character for which `isalpha()` or `isdigit()` is true.

Returns:

Nonzero if `c` is a letter or decimal digit; otherwise, zero.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
{
    if( isalnum( getchar() ) ) {
        printf( "That’s alpha-numeric!\n" );
    }

    return EXIT_SUCCESS;
}
```
isalnum()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The result is valid only for char arguments and EOF.

See also:

isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
isalpha()

Test a character to see if it's alphabetic

Synopsis:

```c
#include <ctype.h>

int isalpha( int c );
```

Arguments:

- `c` The character you want to test.

Library:

```c
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `isalpha()` function tests if the argument `c` is an alphabetic character (`a` to `z` and `A` to `Z`). An alphabetic character is any character for which `isupper()` or `islower()` is true.

Returns:

Nonzero if `c` is an alphabetic character; otherwise, zero.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
{
    if( isalpha( getchar() ) ) {
        printf( "That's alphabetic\n" );
    }

    return EXIT_SUCCESS;
}
```
isalpha()  

Classification:
ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
isalnum(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
Test a character to see if it’s a 7-bit ASCII character

Synopsis:

```c
#include <cctype.h>

int isascii( int c );
```

Arguments:

- `c` The character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `isascii()` function tests for an ASCII character (in the range 0 to 127).

Returns:

Nonzero if `c` is an ASCII character; otherwise, zero.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <cctype.h>

char the_chars[] = { 'A', 0x80, 'Z' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ )
    {
        if( isascii( the_chars[i] ) )
        {
            printf( "Char %c is an ASCII character\n", the_chars[i] );
        }
        else
        {
            printf( "Char %c is not an ASCII character\n", the_chars[i] );
        }
    }
}
```
```c
the_chars[i] );
}
}

return EXIT_SUCCESS;
}

produces the output:

Char A is an ASCII character
Char is not an ASCII character
Char Z is an ASCII character

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalpha(), isalnum(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
**isatty()**

Test to see if a file descriptor is associated with a terminal

**Synopsis:**

```c
#include <unistd.h>

int isatty(int fd);
```

**Arguments:**

`fd`  
The file descriptor that you want to test.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `isatty()` function allows the calling process to determine if the file descriptor `fd` is associated with a terminal.

**Returns:**

- `0`  
The `fd` file descriptor doesn’t refer to a terminal.
- `1`  
The `fd` file descriptor refers to a terminal.

**Examples:**

```c
/*
 * The following program exits with a status of
 * EXIT_SUCCESS if stderr is a tty; otherwise,
 * EXIT_FAILURE
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    return( isatty( 3 ) ? EXIT_SUCCESS : EXIT_FAILURE );
}
```
isatty()

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

open()
iscntrl()

Test a character to see if it's a control character

Synopsis:

```c
#include <ctype.h>

int iscntrl( int c );
```

Arguments:

- **c**  
The character you want to test.

Library:

```c
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `iscntrl()` function tests for any control character. An ASCII control character is any character whose value is between 0 and 31.

Returns:

Nonzero if `c` is a control character; otherwise, zero.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, 'Z' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( iscntrl( the_chars[i] ) ) {
            printf( "Char %c is a Control character\n", the_chars[i] );
        } else {
            printf( "Char %c is not a Control character\n", the_chars[i] );
        }
    }
}
```
iscntrl() produces the output:

Char A is not a Control character
Char is a Control character
Char Z is not a Control character

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
isdigit()

Test a character to see if it’s a decimal digit

Synopsis:

```c
#include <ctype.h>

int isdigit( int c );
```

Arguments:

- `c` The character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `isdigit()` function tests for a decimal digit (characters 0 through 9).

Returns:

Nonzero if `c` is a decimal digit; otherwise, zero.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', '5', '$' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isdigit( the_chars[i] ) ) {
            printf( "Char %c is a digit character\n", the_chars[i] );
        } else {
            printf( "Char %c is not a digit character\n", the_chars[i] );
        }
    }
}
```
isdigit()

```c
    the_chars[i] );
  }
}

return EXIT_SUCCESS;
}

produces the output:

Char A is not a digit character
Char 5 is a digit character
Char $ is not a digit character

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`isalnum()`, `isalpha()`, `iscntrl()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`,
`isspace()`, `isupper()`, `isxdigit()`, `tolower()`, `toupper()`
isfdtype()

Determine whether a file descriptor refers to a socket

Synopsis:

```c
#include <sys/stat.h>

int isfdtype( int filedes,
              int fdtype );
```

Arguments:

- `fdtype` The properties you want to test for. The valid values for `fdtype` include:
  - `S_IFSOCK` — test whether `filedes` is a socket.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `isfdtype()` function determines whether the file descriptor `filedes` has the properties identified by `fdtype`.

This function is based on a POSIX draft; for better portability, call `fstat()` and check the buffer that it fills in:

```c
if (((buf.st_mode & S_IFMT) == fdtype)
    /* The file descriptor matches fdtype. */
else
    /* The file descriptor doesn’t match fdtype. */
```

instead of calling `isfdtype()`.
isfdtype() © 2007, QNX Software Systems GmbH & Co. KG.

Returns:

1 The file descriptor matches fdtype.
0 The file descriptor doesn’t match fdtype.
-1 An error occurred (errno is set).

Errors:

EBADF Invalid file descriptor filedes.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

fstat(), isatty(), socket(), stat()
**Synopsis:**

```c
#include <ctype.h>

int isgraph(int c);
```

**Arguments:**

- `c` The character you want to test.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `isgraph()` function tests for any printable character except a space (`' '`). The `isprint()` function is similar, except that the space character is also included.

**Returns:**

Nonzero if `c` is a printable character (except a space); otherwise, zero.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d };  
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )  
{  
    int i;

    for( i = 0; i < SIZE; i++ ) {  
        if( isgraph( the_chars[i] ) ) {  
            printf( "Char %c is a printable character\n",  
                    the_chars[i] );
        }
    }
}
```
isgraph() produces the output:

Char A is a printable character
Char is not a printable character
Char is not a printable character
Char } is a printable character

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
Synopsis:
#include <math.h>

int isinf ( double x );
int isnff ( float x );

Arguments:
x The number that you want to test.

Library:

libm
Use the -l m option to qcc to link against this library.

Description:
The isinf() and isnff() functions test to see if a number is “infinity.”

Returns:

1 The value of x is infinity.
≠ 1 The value of x isn’t infinity.

Examples:
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;
    a = 2;
    b = -0.5;
    c = NAN;
    fp_exception_mask(_FP_EXC_DIVZERO, 1);
    d = 1.0/0.0;
isinf(), isnff()

```c
printf("%f is %s \n", a, (isinf(a)) ? "infinite" : "not infinite");
printf("%f is %s \n", b, (isinf(b)) ? "infinite" : "not infinite");
printf("%f is %s \n", c, (isinf(c)) ? "infinite" : "not infinite");
printf("%f is %s \n", d, (isinf(d)) ? "infinite" : "not infinite");

return(0);
```

produces the output:

```
2.000000 is not infinite
-0.500000 is not infinite
NAN is not infinite
Inf is infinite
```

**Classification:**

isinf() is ANSI, POSIX 1003.1; isnff() is Unix

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

finite(), isnan()
Synopsis:

#include <ctype.h>

int islower( int c );

Arguments:

$c$ The character you want to test.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The islower() function tests for any lowercase letter $a$ through $z$.

Returns:

Nonzero if $c$ is a lowercase letter; otherwise, zero.

Examples:

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 'a', 'z', 'Z' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ )
    {
        if( islower( the_chars[i] ) )
        {
            printf( "Char %c is a lowercase character\n", the_chars[i] );
        }
        else
        {
            printf( "Char %c is not a lowercase character\n", the_chars[i] );
        }
    }
islower() produces the output:

Char A is not a lowercase character
Char a is a lowercase character
Char z is a lowercase character
Char Z is not a lowercase character

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
Synopsis:

```c
#include <math.h>

int isnan ( double x );
int isnanf ( float x );
```

Arguments:

- `x`: The number you want to test.

Library:

`libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `isnan()` and `isnanf()` functions determine if `x` is Not-A-Number (NAN).

Returns:

- `1`: The value of `x` is NAN.
- `≠ 1`: The value of `x` is a number.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;

    a = 2;
    b = -0.5;
    c = NAN;
```
isnan(), isnanf()

```c
fp_exception_mask(_FP_EXC_DIVZERO, 1);
d = 1.0/0.0;
printf("%f is %s \n", a,
(isnan(a)) ? "not a number" : "a number");
printf("%f is %s \n", b,
(isnan(b)) ? "not a number" : "a number");
printf("%f is %s \n", c,
(isnan(c)) ? "not a number" : "a number");
printf("%f is %s \n", d,
(isnan(d)) ? "not a number" : "a number");
return EXIT_SUCCESS;
```

produces the output:

```
2.000000 is a number
-0.500000 is a number
NAN is not a number
Inf is a number
```

**Classification:**

`isnan()` is POSIX 1003.1; `isnanf()` is Unix

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`finite()`, `isinf()`
Synopsis:

```c
#include <ctype.h>

int isprint(int c);
```

Arguments:

c The character you want to test.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `isprint()` function tests for any printable character, including a space (`' '`). The `isgraph()` function is similar, except that the space character is excluded from the character set being tested.

Returns:

Nonzero if `c` is a printable character; otherwise, zero.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isprint( the_chars[ i ] ) ) {
            printf( "Char %c is a printable character\n", the_chars[ i ] );
        }
    }
}
```


```c
} else {
    printf( "Char %c is not a printable character\n", 
       the_chars[i] );
}
}

return EXIT_SUCCESS;
}

produces the output:

Char A is a printable character
Char is not a printable character
Char is a printable character
Char } is a printable character

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
ispunct()<br>
Test a character to see if it’s any punctuation character

Synopsis:
```
#include <ctype.h>

int ispunct( int c );
```

Arguments:

- `c` The character you want to test.

Library:

```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ispunct()` function tests for any punctuation character such as a comma (,) or a period (.)

Returns:

Nonzero if `c` is punctuation; otherwise, zero.

Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

cchar the_chars[] = { 'A', '!', '.', ',', ':', ';' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( ispunct( the_chars[i] ) ) {
            printf( "Char %c is a punctuation character\n", the_chars[i] );
        } else {
            printf( "Char %c is not a punctuation character\n", the_chars[i] );
        }
    }
}
```
ispunct()

    the_chars[i] );
    }
    }
    return EXIT_SUCCESS;
    }

produces the output:

Char A is not a punctuation character
Char ! is a punctuation character
Char . is a punctuation character
Char , is a punctuation character
Char : is a punctuation character
Char ; is a punctuation character

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), isspace(), isupper(), isxdigit(), tolower(), toupper()
Test a character to see if it’s a whitespace character

Synopsis:

```c
#include <ctype.h>

int isspace( int c );
```

Arguments:

- `c` The character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `isspace()` function tests for the following whitespace characters:

- `' '` space
- `'\f'` form feed
- `'\n'` newline or linefeed
- `'\r'` carriage return
- `'\t'` horizontal tab
- `'\v'` vertical tab

Returns:

Nonzero if `c` is a whitespace character; otherwise, zero.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', 0x09, ' ', 0x7d };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isspace( the_chars[ i ] ) ) {
            printf( "Char %c is a space character\n", the_chars[ i ] );
        } else {
            printf( "Char %c is not a space character\n", the_chars[ i ] );
        }
    }

    return EXIT_SUCCESS;
}
```

This program produces the output:

Char A is not a space character
Char is a space character
Char is a space character
Char } is not a space character

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

iskeanum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isupper(), isxdigit(), tolower(), toupper()
Test a character to see if it’s an uppercase letter

**Synopsis:**

```c
#include <ctype.h>

int isupper(int c);
```

**Arguments:**

- `c` The character you want to test.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `isupper()` function tests for any uppercase letter `A` through `Z`.

**Returns:**

Nonzero if `c` is an uppercase letter; otherwise, zero.

**Examples:**

```c
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

char the_chars[] = { 'A', 'a', 'Z', 'z' };

#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;
    for( i = 0; i < SIZE; i++ ) {
        if( isupper( the_chars[i] ) ) {
            printf( "Char %c is an uppercase character\n", the_chars[i] );
        } else {
            printf( "Char %c is not an uppercase character\n", the_chars[i] );
        }
    }
}
```
isupper()

} } 

return EXIT_SUCCESS; }

produces the output:

Char A is an uppercase character
Char a is not an uppercase character
Char z is not an uppercase character
Char Z is an uppercase character

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isxdigit(), tolower(), toupper()


**iswalnum()**

*Test a wide character to see if it's alphanumeric*

**Synopsis:**

```c
#include <wctype.h>

int iswalnum( wint_t wc );
```

**Arguments:**

- `wc` The wide character you want to test.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iswalnum()` function tests if the argument `wc` is an alphanumeric wide character of the class `alpha` or `digit`. In the C locale, they’re `a` to `z`, `A` to `Z`, `0` to `9`.

**Returns:**

A nonzero value if the character is a member of the class `alpha` or `digit`, or 0 otherwise.

**Classification:**

- ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
iswalnum()

Caveats:

The result is valid only for \texttt{wchar_t} arguments and WEOF.

See also:

\texttt{setlocale()}

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
iswalpha()  © 2007, QNX Software Systems GmbH & Co. KG.

Test a wide character to see if it’s alphabetic

Synopsis:

```
#include <wctype.h>

int iswalpha( wint_t wc );
```

Arguments:

- `wc` The wide character you want to test.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswalpha()` function tests if the argument `wc` is an alphabetic wide character of the class `alpha`. In the C locale, they are: `a` to `z`, `A` to `Z`.

Returns:

A nonzero value if the character is a member of the class `alpha`, or 0 otherwise.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1466  C Library — I to L  September 10, 2007
Caveats:

The result is valid only for `wchar_t` arguments and WEOF.

See also:

`setlocale()`

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Test a wide character to see if it’s a control character

Synopsis:

```c
#include <wctype.h>

int iswcntrl( wint_t wc );
```

Arguments:

- `wc` The wide character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswcntrl()` function tests if the argument `wc` is a control wide character of the class `cntrl`. In the C locale, this class consists of the ASCII characters from 0 through 31.

Returns:

A nonzero value if the character is a member of the class `cntrl`, or 0 otherwise.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The result is valid only for `wchar_t` arguments and WEOF.

See also:

`setlocale()`

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**iswctype()**

Test a wide character to see if it's a given character class

**Synopsis:**

```c
#include <wctype.h>

int iswctype( wint_t wc, wctype_t charclass );
```

**Arguments:**

- `wc` The wide character you want to test.
- `charclass` The character class you want to test for. Get this class by calling `wctype()`.

**Library:**

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `iswctype()` function tests if the argument `wc` is a member of one or several character classes.

<table>
<thead>
<tr>
<th>This function</th>
<th>Is equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iswalnum( wc )</code></td>
<td><code>iswctype( wc , wctype( &quot;alnum&quot; ) )</code></td>
</tr>
<tr>
<td><code>iswalpha( wc )</code></td>
<td><code>iswctype( wc , wctype( &quot;alpha&quot; ) )</code></td>
</tr>
<tr>
<td><code>ispunct( wc )</code></td>
<td><code>iswctype( wc , wctype( &quot;punct&quot; ) )</code></td>
</tr>
</tbody>
</table>
The results are unreliable if you didn’t use \textit{wctype()} to obtain \textit{charclass}, or if a call to \textit{setlocale()} affects LC\_CTYPE.

Returns:

A nonzero value if the character is a member of the specified character class (or classes), or zero if the character isn’t a member or \textit{charclass} is 0.

Classification:

ANSI, POSIX 1003.1

\textbf{Safety}

\begin{tabular}{ll}
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}

Caveats:

The result is valid only for \texttt{wchar_t} arguments and WEOF.

See also:

\textit{setlocale()} 

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**iswdigit()**

Test a wide character to see if it's a decimal digit

**Synopsis:**

```c
#include <wctype.h>

int iswdigit( wint_t wc );
```

**Arguments:**

- `wc` The wide character that you want to test.

**Library:**

`libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iswdigit()` function tests if the argument `wc` is a decimal digit wide character of the class `digit`. In the C locale, this class consists of the characters `0` through `9`.

**Returns:**

A nonzero value if the character is a member of the class `digit`, or 0 otherwise.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
iswdigit()

Caveats:

The result is valid only for \texttt{wchar\_t} arguments and WEOF.

See also:

\texttt{setlocale()}

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Test a wide character to see if it’s any printable character except space

Synopsis:

```
#include <wctype.h>

int iswgraph( wint_t wc );
```

Arguments:

`wc` The wide character you want to test.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswgraph()` function tests if the argument `wc` is a graphical wide character of the class `graph`. In the C locale, this class consists of all the printable characters, except the space character.

Returns:

A nonzero value if the character is a member of the class `graph`, or 0 otherwise.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The result is valid only for \texttt{wchar_t} arguments and WEOF.

See also:

\texttt{setlocale()}

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**iswlower()**

Test a wide character to see if it's a lowercase letter

**Synopsis:**
```
#include <wctype.h>

int iswlower( wint_t wc );
```

**Arguments:**

`wc` The wide character you want to test.

**Library:**
`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iswlower()` function tests if the argument `wc` is a lowercase wide character of the class `lower`. In the C locale, this class consists of the characters from `a` through `z`.

**Returns:**

A nonzero value if the character is a member of the class `lower`, or 0 otherwise.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The result is valid only for \texttt{wchar_t} arguments and WEOF.

See also:

\texttt{setlocale()} 

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Test a wide character to see if it’s any printable character, including space

**Synopsis:**
```
#include <wctype.h>

int iswprint( wint_t wc );
```

**Arguments:**
*wc*  The wide character you want to test.

**Library:**
*libc*
Use the `-l c` option to *qcc* to link against this library. This library is usually included automatically.

**Description:**
The *iswprint()* function tests if the argument *wc* is a printable wide character of the class *print*. In the C locale, this class consists of all the printable characters, including the space character.

**Returns:**
A nonzero value if the character is a member of the class *print*, or 0 otherwise.

**Classification:**
ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:
The result is valid only for \texttt{wchar_t} arguments and WEOF.

See also:
\begin{verbatim}
setlocale()
\end{verbatim}

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
iswpunct()

Test a wide character to see if it’s any punctuation character

Synopsis:

```c
#include <wctype.h>

int iswpunct( wint_t wc );
```

Arguments:

- `wc` The wide character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswpunct()` function tests if the argument `wc` is a punctuation wide character of the class `punct`. In the C locale, this class includes the comma (,) and the period (.), among others.

Returns:

A nonzero value if the character is a member of the class `punct`, or 0 otherwise.

Classification:

- ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**Caveats:**

The result is valid only for `wchar_t` arguments and WEOF.

**See also:**

`setlocale()`

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Test a wide character to see if it's a whitespace character

**Synopsis:**

```c
#include <wctype.h>

int iswspace( wint_t wc );
```

**Arguments:**

- `wc` The wide character you want to test.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iswspace()` function tests if the argument `wc` is a whitespace wide character of the class `space`. In the C locale, this class includes the space character, `\f` (form feed), `\n` (newline or linefeed), `\r` (carriage return), `\t` (horizontal tab), and `\v` (vertical tab).

**Returns:**

A nonzero value if the character is a member of the class `space`, or 0 otherwise.

**Classification:**

ANSI, POSIX 1003.1

---

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued*
iswspace()

Safety

| Thread | Yes |

Caveats:

The result is valid only for wchar_t arguments and WEOF.

See also:

setlocale()

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**iswupper()**

Test a character to see if it’s an uppercase letter

**Synopsis:**

```c
#include <wctype.h>

int iswupper( wint_t wc );
```

**Arguments:**

- `wc` The wide character you want to test.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `iswupper()` function tests if the argument `wc` is an uppercase wide character of the class `upper`. In the C locale, this class includes the characters from `A` through `Z`.

**Returns:**

A nonzero value if the character is a member of the class `upper`, or 0 otherwise.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The result is valid only for `wchar_t` arguments and WEOF.

See also:

`setlocale()`

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Test a wide character to see if it’s a hexadecimal digit

Synopsis:

```c
#include <wctype.h>

int iswxdigit( wint_t wc );
```

Arguments:

- `wc` The wide character you want to test.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `iswxdigit()` function tests if the argument `wc` is a hexadecimal wide character of the class `xdigit`. In the C locale, this class includes the characters 0 to 9, and A to F.

Returns:

A nonzero value if the character is a member of the class `xdigit`, or 0 otherwise.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The result is valid only for `wchar_t` arguments and WEOF.

See also:

`setlocale()`

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
isxdigit() © 2007, QNX Software Systems GmbH & Co. KG.

Test a character to see if it’s a hexadecimal digit

Synopsis:

```
#include <ctype.h>

int isxdigit( int c );
```

Arguments:

c The character you want to test.

Library:

```
libc
```

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The isxdigit() function tests for any hexadecimal-digit character. These characters are the digits 0 through 9 and the letters a through f (or A through F).

Returns:

Nonzero if c is a hexadecimal digit; otherwise, zero.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char the_chars[] = { 'A', '5', '$' };
#define SIZE sizeof( the_chars ) / sizeof( char )

int main( void )
{
    int i;

    for( i = 0; i < SIZE; i++ ) {
        if( isxdigit( the_chars[i] ) ) {
            printf("Char \%c is a hexadecimal digit",
                   the_chars[i]);
        }
    }
}
```


```c
} else {
    printf( "Char %c is not a hexadecimal digit",
            the_chars[i] );
}

return EXIT_SUCCESS;
}
```

produces the output:

Char \texttt{A} is a hexadecimal digit character
Char \texttt{5} is a hexadecimal digit character
Char \texttt{$} is not a hexadecimal digit character

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `tolower()`, `toupper()`
Convert an integer into a string, using a given base

Synopsis:

```c
#include <stdlib.h>

char* itoa( int value, char* buffer, int radix );
```

Arguments:

- `value`: The value to convert into a string.
- `buffer`: A buffer in which the function stores the string. The size of the buffer must be at least:
  
  \[ 8 \times \text{sizeof}(\text{int}) + 1 \]
  
  bytes when converting values in base 2 (binary).
- `radix`: The base to use when converting the number.
  
  If the value of `radix` is 10, and `value` is negative, then a minus sign is prepended to the result.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `itoa()` function converts the integer `value` into the equivalent string in base `radix` notation, storing the result in the specified `buffer`. The function terminates the string with a NUL character.

Returns:

A pointer to the resulting string.
Examples:
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  char buffer[20];
  int base;

  for( base = 2; base <= 16; base += 2 ) {
    printf( "%2d %s\n", base,
            itoa( 12765, buffer, base ) );
  }

  return EXIT_SUCCESS;
}

produces the output:

  2 11000111011101
  4 3013131
  6 135033
  8 30735
 10 12765
 12 7479
 14 491b
 16 31dd

Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

`atoi()`, `atol()`, `itoa()`, `sscanf()`, `strtol()`, `strtoul()`, `ultoa()`, `utoa()`
Synopsis:

```c
#include <math.h>
double j0( double x );
float j0f( float x );
```

Arguments:

- `x` The number that you want to compute the Bessel function for.

Library:

`libbessel`

Use the `-l bessel` option to `qcc` to link against this library.

Description:

Compute the Bessel function of the first kind for `x`.

Returns:

The result of the Bessel function of `x`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;
    x = j0( 2.4 );
    y = y1( 1.58 );
```
\[ z = jn(3, 2.4); \]

\[
\text{printf( "j0(2.4) = %f, y1(1.58) = %f\n", x, y );}
\]

\[
\text{printf( "jn(3,2.4) = %f\n", z );}
\]

return EXIT_SUCCESS;

\}

Classification:

\emph{j0()} is POSIX 1003.1 XSI; \emph{j0f()} is Unix

\begin{center}
\begin{tabular}{ll}
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\hline
\end{tabular}
\end{center}

See also:

\emph{errno, j1(), jn(), y0(), y1(), yn()}
**j1(), j1f()**

*Compute a Bessel function of the first kind*

**Synopsis:**

```c
#include <math.h>

double j1( double x );

float j1f( float x );
```

**Arguments:**

`x`  The number that you want to compute the Bessel function for.

**Library:**

`libbessel`

Use the `-l bessel` option to `qcc` to link against this library.

**Description:**

Compute the Bessel function of the first kind for `x`.

**Returns:**

The result of the Bessel function of `x`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Classification:**

`j1()` is POSIX 1003.1 XSI; `j1f()` is Unix

- **Safety**
  - Cancellation point: No

  *continued...*
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*errno, j0(), jn(), y0(), y1(), yn()*
Compute a Bessel function of the first kind

Synopsis:

```c
#include <math.h>

double jn( int n, double x );

float jnf( int n, float x );
```

Arguments:

- `n, x` The numbers that you want to compute the Bessel function for.

Library:

- `libbessel`
  Use the `-l bessel` option to `qcc` to link against this library.

Description:

Compute the Bessel function of the first kind for `n` and `x`.

Returns:

The result of the Bessel function of `n` and `x`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;

    x = j0( 2.4 );
```
\begin{verbatim}
    y = y1(1.58);
    z = jn(3,2.4);

    printf("j0(2.4) = %f, y1(1.58) = %f\n", x, y);
    printf("jn(3,2.4) = %f\n", z);

    return EXIT_SUCCESS;
\end{verbatim}

**Classification:**

\textit{jn()} is POSIX 1003.1 XSI; \textit{jnf()} is Unix

**Safety**

\begin{tabular}{ll}
  Cancellation point & No \\
  Interrupt handler & No \\
  Signal handler & No \\
  Thread & Yes \\
\end{tabular}

**See also:**

\textit{errno, j0(), j1(), y0(), y1(), yn()}
Generate a pseudo-random signed long integer in a thread-safe manner

Synopsis:

```c
#include <stdlib.h>

long jrand48( unsigned short xsubi[3] );
```

Arguments:

- **xsubi**: An array that comprises the 48 bits of the initial value that you want to use.

Library:

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `jrand48()` function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a signed long integer uniformly distributed over the interval \([-2^{31}, 2^{31})\). It’s a thread-safe version of `mrand48()`.

The `xsubi` array should contain the desired initial value; this makes `jrand48()` thread-safe, and lets you start a sequence of random numbers at any known value.

Returns:

A pseudo-random long integer.

Classification:

- **POSIX 1003.1 XSI**

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>continued...</td>
<td></td>
</tr>
</tbody>
</table>
### jrand48()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`drand48()`, `erand48()`, `lcong48()`, `lrand48()`, `mrand48()`, `nrand48()`, `seed48()`, `srand48()`
kill()

Send a signal to a process or a group of processes

Synopsis:

```c
#include <sys/types.h>
#include <signal.h>

int kill( pid_t pid,
          int sig );
```

Arguments:

- `pid` The ID of the process or process group that you want to send a signal to:
  
  **If `pid` is:**  **Then `sig` is sent to:**
  
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>The single process with that process ID</td>
</tr>
<tr>
<td>0</td>
<td>All processes that are in the same process group as the sending process</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Every process that’s a member of the process group <code>-pid</code></td>
</tr>
</tbody>
</table>

- `sig` Zero, or the signal that you want to send. For a complete list of signals, see “POSIX signals” in the documentation for `SignalAction()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `kill()` function sends the signal `sig` to a process or group of processes specified by `pid`. If `sig` is zero, no signal is sent, but the `pid` is still checked for validity.
For a process to have permission to send a signal to a process, the real or effective user ID of the sending process must either:

- match the real or effective user ID of the receiving process
  
  Or:

- equal zero.

If the value of \textit{pid} causes \textit{sig} to be generated for the sending process, and if \textit{sig} isn’t blocked, either \textit{sig} or at least one pending unblocked signal is delivered before the \textit{kill} function returns.

This call doesn’t block. However, in the network case, lower priority threads may run.

**Returns:**

Zero, or -1 if an error occurs (\textit{errno} is set).

**Errors:**

- \textbf{EAGAIN} Insufficient system resources are available to deliver the signal.
- \textbf{EINVAL} The \textit{sig} is invalid.
- \textbf{EPERM} The process doesn’t have permission to send this signal to any receiving process.
- \textbf{ESRCH} The given \textit{pid} doesn’t exist.

**Examples:**

See \textit{sigprocmask()}. 

**Classification:**

POSIX 1003.1
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`getpid()`, `killpg()`, `setsid()`, `sigaction()`, `signal()`, `SignalKill()`, `sigqueue()`
Send a signal to a process group

**Synopsis:**
```c
#include <sys/types.h>
#include <signal.h>

int killpg( pid_t pgrp,
            int sig );
```

**Arguments:**
- `pid` The ID of the process group that you want to send a signal to.
- `sig` Zero, or the signal that you want to send. For a complete list of signals, see “POSIX signals” in the documentation for SignalAction().

**Library:**
`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `killpg()` function sends the signal `sig` to the process group specified by `pgrp`. If `sig` is zero, no signal is sent, but `pgrp` is still checked for validity.

If `pgrp` is greater than 1, `killpg (pgrp, sig)` is equivalent to `kill (−pgrp, sig)`.

**Returns:**
Zero, or -1 if an error occurs (`errno` is set).

**Errors:**
- `EAGAIN` Insufficient system resources are available to deliver the signal.
- `EINVAL` The signal `sig` is invalid or not supported.
**killpg()**

EPERM The process doesn’t have permission to send this signal to any receiving process.

ESRCH No process group can be found corresponding to the specified `pgrp` or `pgrp` is less than or equal to 1.

**Examples:**

See `sigprocmask()`.

**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`getpid()`, `kill()`, `setsid()`, `sigaction()`, `signal()`, `SignalKill()`, `sigqueue()`
Calculate the absolute value of a long integer

Synopsis:

```
#include <stdlib.h>

long labs( long j );
```

Arguments:

\( j \)  

The number you want the absolute value of.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `labs()` function returns the absolute value of its long-integer argument \( j \).

Returns:

The absolute value of \( j \).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    long x, y;
    x = -50000;
    y = labs( x );
    printf( "labs( %d ) = %d\n", x, y );
    return EXIT_SUCCESS;
}
```

produces the output:

```
labs( -50000 ) = 50000
```
labs()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

abs(), cabs(), fabs()
lc.hown() Change the user ID and group ID of a file or symbolic link

Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>

int lc.hown( const char * path,
             uid_t owner,
             gid_t group );
```

Arguments:

- `path` The name of the file whose ownership you want to change.
- `owner` The user ID of the new owner.
- `group` The group ID of the new owner.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `lc.hown()` function changes the user ID and group ID of the file specified by `path` to be the numeric values contained in `owner` and `group`, respectively. It’s similar to `chown()`, except in the case where the named file is a symbolic link. In this case, `lc.hown()` changes the ownership of the symbolic link file itself, while `chown()` changes the ownership of the file or directory to which the symbolic link refers.

Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges (for example, the superuser) may change the ownership of a file.

In QNX Neutrino, the `_POSIX_CHOWN_RESTRICTED` flag is enforced. This means that only the superuser may change the ownership of a file. The group of a file may be changed by the superuser, or also by a process with the effective user ID equal to the
user ID of the file, if (and only if) owner is equal to the user ID of the
file and group is equal to the effective group ID of the calling process.
If the path argument refers to a regular file, the set-user-ID (S_ISUID) and
set-group-ID (S_ISGID) bits of the file mode are cleared, if the
function is successful.
If lchown() succeeds, the st_ctime field of the file is marked for
update.

Returns:

0 Success.
-1 An error occurred (errno is set).

Errors:

EACCES Search permission is denied on a component of the
path prefix.
ELOOP Too many levels of symbolic links or prefixes.
ENAMETOOLONG
The length of the path string exceeds PATH_MAX, or a
pathname component is longer than NAME_MAX.
ENOENT A component of the path prefix doesn’t exist, or the
path arguments points to an empty string.
ENOSYS The lchown() function isn’t implemented for the
filesystem specified in path.
ENOTDIR A component of the path prefix isn’t a directory.
EPERM The effective user ID does not match the owner of the
file, or the calling process does not have appropriate
privileges.
EROFS The named file resides on a read-only filesystem.
Examples:

/*
   * Change the ownership of a list of files
   * to the current user/group
   */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( lchown( argv[i], getuid(), getgid() ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }
    return( ecode );
}

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chmod(), chown(), errno, fchown(), fstat(), open(), stat()
Synopsis:

```c
#include <stdlib.h>

void lcong48( unsigned short int param[7] );
```

Arguments:

`param` - An array of 7 short integers that are used to initialize the sequence:

- The first three entries are used to initialize the seed.
- The next three are used to initialize the multiplicand.
- The last entry is used to initialize the addend. You can’t use values greater than `0xFFFF` as the addend.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `lcong48()` function gives you full control over the multiplicand and addend used in `drand48()`, `erand48()`, `lrand48()`, `nrand48()`, `mrand48()`, and `jrand48()`, and the seed used in `drand48()`, `lrand48()`, and `mrand48()`.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
See also:

\[ \text{drand48()}, \text{erand48()}, \text{jrand48()}, \text{lrand48()}, \text{mrand48()}, \text{nrand48()}, \text{seed48()}, \text{srand48()} \]
ldexp(), ldexpf()

Multiply a floating-point number by an integral power of 2

Synopsis:

```c
#include <math.h>

double ldexp( double x,
              int exp );

float ldexp( float x,
            int exp );
```

Arguments:

- **x**: A floating-point number.
- **exp**: The exponent of 2 to multiply `x` by.

Library:

- **libm**

Use the `-lm` option to `qcc` to link against this library.

Description:

These functions multiply the floating-point number `x` by \(2^{exp}\).

A range error may occur.

Returns:

\(x \times 2^{exp}\)

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    double value;

    value = ldexp( 4.7072345, 5 );
    printf( "%f\n", value );

    return EXIT_SUCCESS;
}
```

produces the output:

```
150.631504
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

frexp(), modf()}
Synopsis:

```c
#include <stdlib.h>

ldiv_t ldiv( long int numer, long int denom );
```

Arguments:

- `numer` The numerator.
- `denom` The denominator.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ldiv()` function calculates the quotient and remainder of:

`numer ÷ denom`

Returns:

A structure of type `ldiv_t` that contains the following members:

```c
typedef struct {
    long int quot; /* quotient */
    long int rem; /* remainder */
} ldiv_t;
```

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

void print_time( long ticks )
{
    ldiv_t sec_ticks;
    ldiv_t min_sec;
```
```c
sec_ticks = ldiv( ticks, 100 );
min_sec = ldiv( sec_ticks.quot, 60 );

printf( "It took %d minutes and %d seconds.\n", 
        min_sec.quot, min_sec.rem );
}

int main( void )
{
    print_time( 86712 );
    return EXIT_SUCCESS;
}

produces the output:

It took 14 minutes and 27 seconds.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`div()`
Synopsis:

```c
#include <search.h>

void * lfind( const void * key,
              const void * base,
              unsigned * num,
              unsigned width,
              int ( * compare)(
              const void * element1,
              const void * element2 ) );
```

Arguments:

- `key` The object to search for.
- `base` A pointer to the first element in the table.
- `num` A pointer to an integer containing the current number of elements in the table.
- `width` The size of an element, in bytes.
- `compare` A pointer to a user-supplied function that `lfind()` calls to compare an array element with the `key`. The arguments to the comparison function are:
  - `element1` — the same pointer as `key`
  - `element2` — a pointer to one of the array elements.

The comparison function must return 0 if `element1` equals `element2`, or a nonzero value if the elements aren’t equal.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The \texttt{lfind()} function returns a pointer into a table indicating where an entry may be found.

The \texttt{lfind()} function is the same as \texttt{lsearch()}, except that if the entry isn’t found, it isn’t added to the table, and a NULL pointer is returned.

Returns:

A pointer to the matching element, or NULL if there’s no match or an error occurred.

Examples:

This example program lets you know if the first command-line argument is a C keyword (assuming you fill in the \textit{keywords} array with a complete list of C keywords):

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <search.h>

static const char *keywords[] = {
  "auto",
  "break",
  "case",
  "char",
  /* . */
  /* . */
  /* . */
  "while"
};

int compare( const void *, const void * );

int main( int argc, const char *argv[] )
{
  unsigned num = 5;
  char *ptr;
  if( argc <= 1 ) return EXIT_FAILURE;
  ptr = lfind( &argv[1], keywords, &num, sizeof(char **), compare );
  if( ptr == NULL ) {
    printf( "%s\n", argv[1] );
    return EXIT_FAILURE;
  } else {
    ...
  }
}
```
printf( "'%s' is a C keyword\n", argv[1] );

    return EXIT_SUCCESS;
}

/* You’ll never get here. */
return EXIT_SUCCESS;
}

int compare( const void *op1, const void *op2 )
{
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;

    return( strcmp( *p1, *p2 ) );
}

Classification:

POSIX 1003.1 XSI

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

*bsearch(), lsearch()
lgamma(), lgamma_r(), lgammaf(), lgammaf_r()

Log gamma function

Synopsis:

```c
#include <math.h>

double lgamma( double x );
double lgamma_r( double x, int* signgamp );
float lgammaf( float x );
float lgammaf_r( float x, int* signgamp );
```

Arguments:

- **x** An arbitrary number.
- **signgam** *(lgamma_r(), lgammaf_r() only) A pointer to a location where the function can store the sign of $\Gamma(x)$.

Library:

```c
libm
```

Use the `-l m` option to `qcc` to link against this library.

Description:

The `lgamma()` and `lgamma_r()` functions return the natural log (\(\ln\)) of the \(\Gamma\) function and are equivalent to `gamma()`. These functions return \(\ln|\Gamma(x)|\), where \(\Gamma(x)\) is defined as follows:

\[
\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt
\]

For \(x > 0:\)

\[\Gamma(x) = \frac{n}{(\Gamma(1-x) \cdot \sin(nx))}\]

The results converge when \(x\) is between 0 and 1. The \(\Gamma\) function has the property:
\[ \Gamma(N) = \Gamma(N-1) \times N \]

The `lgamma*` functions compute the log because the \( \Gamma \) function grows very quickly.

The `lgamma()` and `lgammaf()` functions use the external integer `signgam` to return the sign of \( \Gamma(x) \), while `lgamma_r()` and `lgammaf_r()` use the user-allocated space addressed by `signgamp`.

The `signgam` variable isn’t set until `lgamma()` or `lgammaf()` returns. For example, don’t use the expression:

\[
g = \text{signgam} \times \exp(\text{lgamma}(x));
\]

to compute \( g = \Gamma(x)' \). Instead, compute `lgamma()` first:

\[
\text{lg} = \text{lgamma}(x);
\text{g} = \text{signgam} \times \exp(\text{lg});
\]

Note that \( \Gamma(x) \) must overflow when \( x \) is large enough, underflow when \( -x \) is large enough, and generate a division by 0 exception at the singularities \( x \) a nonpositive integer.

**Returns:**

\[ \ln|\Gamma(x)| \]

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv) {
```
double a, b;

erno = EOK;
a = 0.5;
b = lgamma(a);
printf("lgamma(%f) = %f %d \n", a, b, errno);

return(0);
}

produces the output:

\texttt{lgamma(0.500000) = 0.572365 0}

**Classification:**

\texttt{lgamma()} and \texttt{lgammaf()} are ANSI, POSIX 1003.1; \texttt{lgamma_r()} and \texttt{lgammaf_r()} are QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

\texttt{gamma()}

\textcopyright{} 2007, QNX Software Systems GmbH & Co. KG.
link()

Create a link to an existing file

Synopsis:

```
#include <unistd.h>

int link( const char* existing, const char* new );
```

Arguments:

- `existing`: The path of an existing file.
- `new`: The path for the new link.

Library:

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `link()` function creates a new directory entry named by `new` to refer to (that is, to be a link to) an existing file named by `existing`. The function atomically creates a new link for the existing file, and increments the link count of the file by one.

This implementation doesn’t support using `link()` on directories or the linking of files across filesystems (different logical disks).

If the function fails, no link is created, and the link count of the file remains unchanged.

If `link()` succeeds, the `st_ctime` field of the file and the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.
Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EACCES A component of either path prefix denies search permission, or the link named by new is in a directory with a mode that denies write permission.

EEXIST The link named by new already exists.

ELOOP Too many levels of symbolic links or prefixes.

EMLINK The number of links to the file named by existing would exceed LINK_MAX.

ENAMETOOLONG

The length of the existing or new string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT This error code can mean the following:

- A component of either path prefix doesn’t exist.
- The file named by existing doesn’t exist.
- Either existing or new points to an empty string.

ENOSPC The directory that would contain the link can’t be extended.

ENOSYS The link() function isn’t implemented for the filesystem specified in existing or new.

ENOTDIR A component of either path prefix isn’t a directory.

EPERM The file named by existing is a directory.

EROFS The requested link requires writing in a directory on a read-only file system.
EXDEV The link named by new and the file named by existing are on different logical disks.

Examples:

```c
/*
 * The following program performs a rename operation of argv[1] to argv[2].
 * Please note that this example, unlike the * library function rename(), ONLY works if
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main( int argc, char** argv )
{
  /* Create a link of argv[1] to argv[2].
   */
  if( link( argv[1], argv[2] ) == -1 ) {
    perror( "link" );
    return( EXIT_FAILURE );
  }
  if( unlink( argv[1] ) == -1 ) {
    perror( argv[1] );
    return( EXIT_FAILURE );
  }
  return( EXIT_SUCCESS );
}
```

Classification:

POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

```
erro, rename(), symlink(), unlink()
```
Synopsis:

```c
#include <aio.h>

int lio_listio( int mode,
    struct aiocb* const list[],
    int nent,
    struct sigevent* sig );
```

Arguments:

- **mode** The mode of operation; one of:
  - LIO_WAIT — `lio_listio()` behaves synchronously, waiting until all I/O is completed, and ignores the `sig` argument.
  - LIO_NOWAIT — `lio_listio()` behaves asynchronously, returning immediately, and the signal specified by the `sig` argument is delivered to the calling process when all the I/O operations from this function complete.

- **list** An array of pointers to `aiocb` structures that specify the I/O operations that you want to initiate. The array may contain NULL pointers, which the function ignores.

- **nent** The number of entries in the `list` array. This must not exceed the system-wide limit, `_POSIX_AIO_MAX`.

- **sig** NULL, or a pointer to a `sigevent` structure that specifies the signal that you want to deliver to the calling process when all of the I/O operations complete. The function ignores this argument if `mode` is LIO_WAIT.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `lio_listio()` function lets the calling process, lightweight process (LWP), or thread initiate a list of I/O requests within a single function call.

The `aio_lio_opcode` field of each `aiocb` structure in `list` specifies the operation to be performed (see `<aio.h>`):

- LIO_READ requests `aio_read()`.
- LIO_WRITE requests `aio_write()`.
- LIO_NOP causes the list entry to be ignored.

If `mode` is LIO_NOWAIT, `lio_listio()` uses the `sigevent` structure pointed to by `sig` to define both the signal to be generated and how the calling process is notified when the I/O operations are complete:

- If `sig` is NULL, or the `sigev_signo` member of the `sigevent` structure is zero, then no signal delivery occurs. Otherwise, the signal number indicated by `sigev_signo` is delivered when all the requests in the list have completed.
- If `sig->sigev_notify` is SIGEV_NONE, no signal is posted upon I/O completion, but the error status and the return status for the operation are set appropriately.
- If `sig->sigev_notify` is SIGEV_SIGNAL, the signal specified in `sig->sigev_signo` is sent to the process. If the SA_SIGINFO flag is set for that signal number, the signal is queued to the process, and the value specified in `sig->sigev_value` is the `si_value` component of the generated signal.

For regular files, no data transfer occurs past the offset maximum established in the open file description associated with `aiocbp->aio_fildes`.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file.
associated with `aio_fildes`. (see the definitions of O_DSYNC and O_SYNC in the description of `fcntl()`.)

Returns:

If the `mode` argument is LIO_NOWAIT, and the I/O operations are successfully queued, `lio_listio()` returns 0; otherwise, it returns -1, and sets `errno`.

If the `mode` argument is LIO_WAIT, and all the indicated I/O has completed successfully, `lio_listio()` returns 0; otherwise, it returns -1, and sets `errno`.

In either case, the return value indicates only the success or failure of the `lio_listio()` call itself, not the status of the individual I/O requests. In some cases, one or more of the I/O requests contained in the list may fail. Failure of an individual request doesn’t prevent completion of any other individual request. To determine the outcome of each I/O request, examine the error status associated with each `aiocb` control block. Each error status so returned is identical to that returned as a result of calling `aio_read()` or `aio_write()`.

Errors:

EAGAIN The resources necessary to queue all the I/O requests weren’t available. The error status for each request is recorded in the `aio_error` member of the corresponding `aiocb` structure, and can be retrieved using `aio_error()`.

The number of entries, `nent`, exceeds the system-wide limit, _POSIX_AIO_MAX.

EINVAL The `mode` argument is invalid.

The value of `nent` is greater than _POSIX_AIO_LISTIO_MAX.

EINTR A signal was delivered while waiting for all I/O requests to complete during an LIO_WAIT operation. However, the outstanding I/O requests aren’t canceled. Use `aio_fsync()` to determine if any request was initiated;
aio_return() to determine if any request has completed; or aio_error() to determine if any request was canceled.

EIO One or more of the individual I/O operations failed. Use aio_error() with each aiocb structure to determine which request(s) failed.

ENOSYS The lio_listio() function isn’t supported by this implementation.

If either lio_listio() succeeds in queuing all of its requests, or errno is set to EAGAIN, EINTR, or EIO, then some of the I/O specified from the list may have been initiated. In this event, each aiocb structure contains errors specific to the read() or write() function being performed:

EAGAIN The requested I/O operation wasn’t queued due to resource limitations.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit aio_cancel() request.

EINPROGRESS The requested I/O is in progress.

The following additional error codes may be set for each aiocb control block:

EOVERFLOW The aiocbp->aio_lio_opcode is LIO_READ, the file is a regular file, aiocbp->aio_nbytes is greater than 0, and the aiocbp->aio_offset is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with aiocbp->aio_fildes.

EFBIG The aiocbp->aio_lio_opcode is LIO_WRITE, the file is a regular file, aiocbp->aio_nbytes is greater than 0, and the aiocbp->aio_offset is greater than or equal to the offset maximum in the open file description associated with aiocbp->aio_fildes.
Classification:

POSIX 1003.1 AIO

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

aio_cancel(), aio_error(), aio_fsync(), aio_read(), aio_return(),
aio_write(), close(), execcl(), execle(), execlp(), execle(), execv(),
execve(), execvp(), execvepe(), exit(), fcntl(), fork(), lseek(), read(),
sigevent.write()
Listen for connections on a socket

Synopsis:

```c
#include <sys/socket.h>

int listen( int s,
            int backlog );
```

Arguments:

- `s` The descriptor for the socket that you want to listen on. You can create a socket by calling `socket()`.
- `backlog` The maximum length that the queue of pending connections may grow to.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `listen()` function listens for connections on a socket and puts the socket into the LISTEN state. For connections to be accepted, you must:

1. Create a socket by calling `socket()`.
2. Indicate a willingness to accept incoming connections and a queue limit for them by calling `listen()`.
3. Call `accept()` to accept the connections.

If a connection request arrives with the queue full, the client may receive an error with an indication of ECONNREFUSED. But if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.
The **listen()** call applies only to SOCK_STREAM sockets.

### Returns:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (<strong>errno</strong> is set).</td>
</tr>
</tbody>
</table>

### Errors:

- **EBADF** Invalid descriptor *s*.
- **EOPNOTSUPP**
  
  The socket isn’t of a type that supports the **listen()** operation.

### Classification:

- POSIX 1003.1

### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

*accept(), connect(), socket()*
localeconv() © 2007, QNX Software Systems GmbH & Co. KG.

Set numeric formatting according to the current locale

Synopsis:

```c
#include <locale.h>

struct lconv * localeconv( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `localeconv()` function gets the values appropriate for formatting numeric quantities using the current locale. It returns a pointer to a `struct lconv` with the following members:

- `char * decimal_point`
  The decimal-point character used for nonmonetary quantities.

- `char * thousands_sep`
  The character used to separate groups of digits on the left of the decimal-point character formatted nonmonetary quantities.

- `char * int_curr_symbol`
  The international currency symbol for the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217: Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the NUL character) is the character used to separate the international currency symbol from the monetary quantity.

- `char * currency_symbol`
  The local currency symbol applicable to the current locale.

- `char * mon_decimal_point`
  The decimal-point character used to format monetary quantities.
localeconv()  

char * mon_thousands_sep
The character used to separate groups of digits on the left of the decimal-point character in formatted monetary quantities.

char * mon_grouping
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char * grouping
A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char * positive_sign
The string used to indicate a nonnegative monetary quantity.

char * negative_sign
The string used to indicate a negative monetary quantity.

char int_frac_digits
The number of fractional digits (to the right of the decimal point) to display in an internationally formatted monetary quantity.

char frac_digits
The number of fractional digits (to the right of the decimal point) to display in a formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the currency_symbol precedes or follows the value for a nonnegative monetary quantity.

char p_sep_by_space
Set to 1 or 0 if the currency_symbol is or isn’t separated by a space from the value for a nonnegative monetary quantity.

char n_cs_precedes
Set to 1 or 0 if the currency_symbol precedes or follows the value for a negative monetary quantity.
localeconv()

char n_sep_by_space
    Set to 1 or 0 if the currency_symbol is or isn’t separated by a space from the value for a negative monetary quantity.

char p_sign_posn
    The position of the positive_sign for a nonnegative monetary quantity.

char n_sign_posn
    The position of the positive_sign for a negative monetary quantity.

The grouping and mon_grouping members have the following values:

CHAR_MAX       Perform no further grouping.
0               Repeat the previous element used for the remainder of the digits.
other           The value is the number of digits that comprise the current group. Examine the next element to determine the size of the next group of digits (to the left of the current group).

The p_sign_posn and n_sign_posn members have the following values:

0               Parentheses surround the quantity and currency_symbol.
1               The sign string precedes the quantity and currency_symbol.
2               The sign string follows the quantity and currency_symbol.
3               The sign string immediately precedes the quantity and currency_symbol.
4               The sign string immediately follows the quantity and currency_symbol.
localeconv()

Returns:

A pointer to the struct lconv.

Examples:

```c
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>

int main( void )
{
    struct lconv *lc;
    lc = localeconv();
    printf( "decimal_point (%s)\n", lc->decimal_point );
    printf( "thousands_sep (%s)\n", lc->thousands_sep );
    printf( "int_curr_symbol (%s)\n", lc->int_curr_symbol );
    printf( "currency_symbol (%s)\n", lc->currency_symbol );
    printf( "mon_decimal_point (%s)\n", lc->mon_decimal_point );
    printf( "mon_thousands_sep (%s)\n", lc->mon_thousands_sep );
    printf( "mon_grouping (%s)\n", lc->mon_grouping );
    printf( "grouping (%s)\n", lc->grouping );
    printf( "positive_sign (%s)\n", lc->positive_sign );
    printf( "negative_sign (%s)\n", lc->negative_sign );
    printf( "int_frac_digits (%d)\n", lc->int_frac_digits );
    printf( "frac_digits (%d)\n", lc->frac_digits );
```
localeconv()

```c
    printf("p_cs_precedes (%d)\n", lc->p_cs_precedes);
    printf("p_sep_by_space (%d)\n", lc->p_sep_by_space);
    printf("n_cs_precedes (%d)\n", lc->n_cs_precedes);
    printf("n_sep_by_space (%d)\n", lc->n_sep_by_space);
    printf("p_sign_posn (%d)\n", lc->p_sign_posn);
    printf("n_sign_posn (%d)\n", lc->n_sign_posn);
    return EXIT_SUCCESS;
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

isalpha(), isascii(), printf(), scanf(), setlocale(), strcat(), strchr(), strcmp(), strcoll(), strcpy(), strfiome(), strlen(), strpbrk(), strspn(), strtod(), strtok(), strxfrm()
localtime()

Convert calendar time to local time

Synopsis:

```c
#include <time.h>

struct tm *localtime( const time_t *timer );
```

Arguments:

- `timer` A pointer to a `time_t` object that contains the calendar time that you want to convert.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `localtime()` function converts the calendar time pointed to by `timer` into local time, storing the information in a `struct tm`. Whenever you call `localtime()`, it calls `tzset()`.

You typically get a calendar time by calling `time()`. That time is Coordinated Universal Time (UTC, formerly known as Greenwich Mean Time or GMT).

The `localtime()` function places the converted time in a static `tm` structure that’s reused each time you call `localtime()` or `gmtime()`. Calling `asctime()` or `ctime()` could also change the data in this static buffer.

In a multithreaded application, use `localtime_r()`.

You typically use the `date` command to set the computer’s internal clock using Coordinated Universal Time (UTC). Use the `TZ` environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.
localtime()

Returns:

A pointer to the static \texttt{tm} structure containing the time information.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), clock(), ctime(), ctime_r(), difftime(),
gmtime(), gmtime_r(), mktime(), localtime_r(), strftime(), time(), \texttt{tm},
tzset()

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User's Guide
localtime_r()

Convert calendar time to local time

Synopsis:

#include <time.h>

struct tm* localtime_r( const time_t* timer,  
                      struct tm* result );

Arguments:

timer A pointer to a time_t object that contains the calendar time that you want to convert.

result A pointer to a tm structure where the function can store the converted time.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The localtime_r() function converts the calendar time pointed to by timer into local time, storing the information in the struct tm that result points to. Whenever you call localtime_r(), it calls tzset().

You typically get a calendar time by calling time(). That time is Coordinated Universal Time (UTC, formerly known as Greenwich Mean Time or GMT).

You typically use the date command to set the computer’s internal clock using Coordinated Universal Time (UTC). Use the TZ environment variable or _CS_TIMEZONE configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.
localtime_r()

Returns:
A pointer to result, the struct tm.

Classification:
POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
asctime(), asctime_r(), clock(), ctime(), ctime_r(), difftime(),
gmtime(), gmtime_r(), localtime(), mktime(), strftime(), time(), tm,
tzset()

“Setting the time zone” in the Configuring Your Environment chapter
of the Neutrino User’s Guide
lockf()

Lock or unlock a section of a file

Synopsis:

```c
#include <unistd.h>

int lockf( int filedes,
           int function,
           off_t size );
```

Arguments:

- **fildes** The file descriptor for the file that you want to lock. Open the file with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

- **function** A control value that specifies the action to be taken. The permissible values (defined in `<unistd.h>`) are as follows:

  - F_LOCK: Lock a section for exclusive use if the section is available. A read-only lock is one of O_RDONLY, O_WRONLY, or O_RDWR. An exclusive lock is one of O_WRONLY, or O_RDWR. (For descriptions of the locks, see `open()`).

  - F_TEST: Test a specified section for locks obtained by other processes.

  - F_TLOCK: Test and lock a section for exclusive use if the section is available.

  - F_UNLOCK: Remove locks from a specified section of the file.

- **size** The number of contiguous bytes that you want to lock or unlock. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive `size` or backward for a negative `size` (the preceding bytes up to but not including the current offset). If `size` is 0, the section from the current offset through the largest possible file offset is locked (that is,
from the current offset through to the present or any future end-of-file). An area need not be allocated to the file to be locked because locks may exist past the end-of-file.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

You can use the `lockf()` function to lock a section of a file, using advisory-mode locks. If other threads call `lockf()` to try to lock the locked file section, those calls either return an error value or block until the section becomes unlocked.

All the locks for a process are removed when the process terminates. Record locking with `lockf()` is supported for regular files and may be supported for other files.

The sections locked with `F_LOCK` or `F_TLOCK` may in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections occur, the sections are combined into a single locked section.

File locks are released on the first close by the locking process of any file descriptor for the file.

`F_ULOCK` requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections are unlocked starting at the current file offset through `size` bytes or to the end of file if `size` is `(off_t)0`. When all of a locked section isn’t released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. Releasing the center portion of a locked section causes the remaining locked beginning and end portions to become two separate locked sections.
lockf()

A potential for deadlock occurs if the threads of a process controlling a locked section are blocked by accessing another process’s locked section. If the system detects that deadlock could occur, lockf() fails with EDEADLK.

The interaction between fcntl() and lockf() locks is unspecified. Blocking on a section is interrupted by any signal.

If size is the maximum value of type off_t and the process has an existing lock of size 0 in this range (indicating a lock on the entire file), then an F_ULOCK request is treated the same as an F_ULOCK request of size 0. Otherwise an F_ULOCK request attempts to unlock only the requested section. Attempting to lock a section of a file that’s associated with a buffered stream produces unspecified results.

Returns:

0 Success.

-1 An error occurred (errno is set). Existing locks aren’t changed.

Errors:

EACCES or EAGAIN
The function argument is F_TLOCK or F_TEST and the section is already locked by another process.

EAGAIN The function argument is F_LOCK or F_TLOCK and the file is mapped with mmap().

EBADF The fildes argument isn’t a valid open file descriptor; or function is F_LOCK or F_TLOCK and fildes isn’t a valid file descriptor open for writing.

EDEADLK The function argument is F_LOCK and a deadlock is detected.

EINTR A signal was caught during execution of the function.
lockf()

The function argument isn’t one of F_LOCK, F_TLOCK, F_TEST or F_ULOCK; or size plus the current file offset is less than 0.

The system can’t allocate sufficient memory to store lock resources.

The implementation doesn’t support the locking of files of the type indicated by fildes.

The offset of the first, or if size isn’t 0 then the last, byte in the requested section can’t be represented correctly in an object of type off_t.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fcntl(), flock(), open()
compute the natural logarithm of a number

Synopsis:

```c
#include <math.h>

double log( double x );

float logf( float x );
```

Arguments:

- `x`: The number that you want to compute the natural log of.

Library:

- `libm`
  
  Use the `-l m` option to `qcc` to link against this library.

Description:

The `log()` and `logf()` functions compute the natural logarithm (base `e`) of `x`:

\[
\log_e x
\]

A domain error occurs if `x` is negative. A range error occurs if `x` is zero.

Returns:

The natural logarithm of `x`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", log(.5) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
-0.693147
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno, exp(), log10(), pow()`
Synopsis:

```
#include <math.h>

double log1p ( double x );

float log1pf ( float x );
```

Arguments:

- \( x \) The number that you want to add 1 to and compute the natural log of.

Library:

- `libm`
  
  Use the `-l m` option to `qcc` to link against this library.

Description:

The `log1p()` and `log1pf()` functions compute the value of \( \log(1+x) \), where \( x > -1.0 \).

Returns:

<table>
<thead>
<tr>
<th>If</th>
<th><code>log1p()</code> returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{NAN} )</td>
<td>\text{NAN}</td>
</tr>
<tr>
<td>( x &lt; -1.0 )</td>
<td>-HUGE_VAL, or \text{NAN} (\text{errno} is set to \text{EDOM}).</td>
</tr>
<tr>
<td>( x = -1.0 )</td>
<td>-HUGE_VAL (\text{errno} may be set to \text{ERANGE}).</td>
</tr>
</tbody>
</table>

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set \( \text{errno} \) to 0, call the function, and then check \( \text{errno} \) again. These functions don’t change \( \text{errno} \) if no errors occurred.
**log1p(), log1pf()**

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

ilogb(), log(), logb(), log10()
log10(), log10f()

Compute the logarithm (base 10) of a number

Synopsis:

#include <math.h>

double log10( double x);

float log10f( float x);

Arguments:

x The number that you want to compute the log of.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The log10() and log10f() functions compute the base 10 logarithm of x:

\log_{10} x

A domain error occurs if x is negative. A range error occurs if x is zero.

Returns:

The base 10 logarithm of x.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set errno to 0, call the function, and then check errno again. These functions don’t change errno if no errors occurred.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%f\n", log10(.5) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
-0.301030
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno, exp(), log(), pow()`
Synopsis:

```
#include <math.h>

double logb ( double x );

float logbf ( float x );
```

Arguments:

- `x` The number that you want to compute the radix-independent exponent of.

Library:

```
libm
```

Use the `-l m` option to `qcc` to link against this library.

Description:

The `logb()` and `logbf()` functions compute the exponent part of `x`, which is the integral part of:

```
log_r |x|
```

as a signed floating point value, for nonzero finite `x`, where `r` is the radix of the machine’s floating point arithmetic.

Returns:

The binary exponent of `x`, a signed integer converted to double-precision floating-point.

```
<table>
<thead>
<tr>
<th>If x is:</th>
<th>logb() returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>-HUGE_VAL (errno is set to EDOM)</td>
</tr>
<tr>
<td>&lt;0.0</td>
<td>-HUGE_VAL (errno may be set to ERANGE)</td>
</tr>
</tbody>
</table>
```

continued...
logb(), logbf()

If \( x \) is:  \quad logb() returns:

| ±infinity | +infinity |

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set \( errno \) to 0, call the function, and then check \( errno \) again. These functions don’t change \( errno \) if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;
    a = 0.5;
    b = logb(a);
    printf("logb(%f) = %f (%f = 2^{%f}) \n", a, b, a, b);
    return(0);
}
```

produces the output:

```
logb(0.500000) = -1.000000 (0.500000 = 2^{-1.000000})
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued…
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`ilogb()`, `log()`, `log10()`, `log1p()`
login_tty()  
Prepare for a login in a tty

Synopsis:

```c
#include <unix.h>

int login_tty( int fd );
```

Arguments:

- `fd` A file descriptor that you want to use as the controlling terminal for the current process.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `login_tty()` function prepares for a login on the tty `fd` (which may be a real tty device, or the slave of a pseudo-tty as returned by `openpty()`) by creating a new session, making `fd` the controlling terminal for the current process, setting `fd` to be the standard input, output, and error streams of the current process, and closing `fd`.

This function fails if `ioctl()` fails to set `fd` to the controlling terminal of the current process.

Returns:

- 0 Success.
- -1 An error occurred; `errno` is set.
login_tty()

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

forkpty(), ioctl(), openpty()
**longjmp()**

Restore the environment saved by setjmp()

### Synopsis:

```c
#include <setjmp.h>

void longjmp(jmp_buf env, int return_value);
```

### Arguments:

- `env` The environment saved by the most recent call to `setjmp()`.
- `return_value` The value that you want `setjmp()` to return.

### Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `longjmp()` function restores the environment saved in `env` by the most recent call to the `setjmp()` function.

Using `longjmp()` to jump out of a signal handler can cause unpredictable behavior, unless the signal was generated by the `raise()` function.

### Returns:

After the `longjmp()` function restores the environment, program execution continues as if the corresponding call to `setjmp()` had just returned the value specified by `return_value`. If the value of `return_value` is 0, the value returned is 1.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;

void rtn( void )
{
    printf( "about to longjmp\n" );
    longjmp( env, 14 );
}

int main( void )
{
    int ret_val = 293;
    if( 0 == ( ret_val = setjmp( env )) ) {
        printf( "after set jmp %d\n", ret_val );
        rtn();
        printf( "back from rtn %d\n", ret_val );
    } else {
        printf( "back from long jmp %d\n", ret_val );
    }

    return EXIT_SUCCESS;
}
```

produces the following output:

```
after set jmp 0
about to long jmp
back from long jmp 14
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued...*
**longjmp()**

**Safety**

| Thread | Yes |

**Caveats:**

A strictly-conforming POSIX application can’t assume that the `longjmp()` function is signal-safe on other platforms.

---

**WARNING:** Don’t use `longjmp()` or `siglongjmp()` to restore an environment saved by a call to `setjmp()` or `sigsetjmp()` in another thread. If you’re lucky, your application will crash; if not, it’ll look as if it works for a while, until random scribbling on the stack causes it to crash.

---

**See also:**

`setjmp()`, `siglongjmp()`, `sigsetjmp()`
Synopsis:

```c
#include <stdlib.h>

long lrand48( void );
```

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `lrand48()` function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative `long` integer uniformly distributed over the interval `[0, 2^{31}]`.

Call one of `lcong48()`, `seed48()`, or `srand48()` to initialize the random-number generator before calling `drand48()`, `lrand48()`, or `mrand48()`.

The `nrand48()` function is a thread-safe version of `lrand48()`.

Returns:

A pseudo-random `long` integer.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

drand48(), erand48(), jrand48(), lcong48(), mrand48(), nrand48(),
seed48(), srand48()
Synopsis:

```c
#include <search.h>

void * lsearch( const void * key,
    const void * base,
    unsigned * num,
    unsigned width,
    int ( * compare)(
        const void * element1,
        const void * element2 ) );
```

Arguments:

- **key**  The object to search for.
- **base**  A pointer to the first element in the table.
- **num**   A pointer to an integer containing the current number of elements in the table.
- **width** The size of an element, in bytes.
- **compare**  A pointer to a user-supplied function that **lsearch()** calls to compare an array element with the **key**. The arguments to the comparison function are:
  - **element1** — the same pointer as **key**
  - **element2** — a pointer to one of the array elements.

The comparison function must return 0 if **element1** equals **element2**, or a nonzero value if the elements aren’t equal.

Library:

**libc**

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.
Description:

The `lsearch()` function searches a linear table and returns a pointer into the table indicating where the entry was found.

If key isn’t found, it’s added to the end of the array and num is incremented.

Returns:

A pointer to the element that was found or created, or NULL if an error occurred.

Examples:

This program builds an array of pointers to the `argv` arguments by searching for them in an array of NULL pointers. Because none of the items will be found, they’ll all be added to the array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <search.h>

int compare( const void *, const void * );

int main( int argc, const char **argv )
{
    int i;
    unsigned num = 0;
    char **array = (char **)calloc( argc, sizeof(char **) );
    for( i = 1; i < argc; ++i ) {
        lsearch( &argv[i], array, &num, sizeof(char **), compare );
    }
    for( i = 0; i < num; ++i ) {
        printf( "%s\n", array[i] );
    }
    return EXIT_SUCCESS;
}

int compare( const void *op1, const void *op2 )
{
    const char **pl = (const char **)op1;
    const char **p2 = (const char **)op2;
    return( strcmp( *pl, *p2 ) );
}
```
Using the program above, this input:
\texttt{one two one three four}

produces the output:

\begin{verbatim}
one
two
three
four
\end{verbatim}

\section*{Classification:}

\begin{itemize}
\item POSIX 1003.1 XSI
\end{itemize}

\begin{center}
\begin{tabular}{l|c}
\textbf{Safety} & \\
Creation point & No \\
Interrupt handler & Yes \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}
\end{center}

\section*{See also:}

\begin{itemize}
\item \texttt{bsearch()}, \texttt{llfind()}
\end{itemize}
**Synopsis:**

```c
#include <sys/types.h>
#include <unistd.h>

off_t lseek( int filedes, 
             off_t offset, 
             int whence );

off64_t lseek64( int filedes, 
                 off64_t offset, 
                 int whence );
```

**Arguments:**

- **filedes**
  The file descriptor of the file whose position you want to set.

- **offset**
  The relative offset from the file position determined by the `whence` argument.

- **whence**
  The position in the file. The possible values (defined in `<unistd.h>`) are:

  - SEEK_CUR
    The new file position is computed relative to the current file position. The value of `offset` may be positive, negative or zero.

  - SEEK_END
    The new file position is computed relative to the end of the file.

  - SEEK_SET
    The new file position is computed relative to the start of the file. The value of `offset` must not be negative.

**Library:**

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

These functions set the current file position for the file descriptor specified by `filedes` at the operating system level. File descriptors are returned by a successful execution of one of the `creat()`, `dup()`, `dup2()`, `fcntl()`, `open()` or `sopen()` functions.

An error occurs if the requested file position is before the start of the file.

If the requested file position is beyond the end of the file and data is written at this point, subsequent reads of data in the gap will return bytes whose value is equal to zero (`\0`) until data is actually written into the gap.

These functions don’t extend the size of a file (see `chsize()`).

Returns:

The current file position, with 0 indicating the start of the file, or -1 if an error occurs (`errno` is set).

Errors:

- **EBADF** The `filedes` argument isn’t a valid file descriptor.
- **EINVAL** The `whence` argument isn’t a proper value, or the resulting file offset is invalid.
- **ENOSYS** The `lseek()` function isn’t implemented for the filesystem specified by `filedes`.
- **EOVERFLOW** The resulting file offset is a value that can’t be represented correctly in an object of type `off_t`.
- **ESPIPE** The `filedes` argument is associated with a pipe or FIFO.
Examples:

Using the `lseek()` function, you can get the current file position (in fact, `tell()` is implemented this way). You can then use this value with another call to `lseek()` to reset the file position:

```c
off_t file_posn;
int filedes;

/* get current file position */
file_posn = lseek( filedes, 0L, SEEK_CUR );
...

/* return to previous file position */
file_posn = lseek( filedes, file_posn, SEEK_SET );
```

If all records in the file are the same size, the position of the \(n\)th record can be calculated and read like this:

```c
#include <sys/types.h>
#include <unistd.h>

int read_record( int filedes, long rec_numb, int rec_size, char *buffer )
{
    if( lseek( filedes, rec_numb * rec_size, SEEK_SET ) == -1L ) {
        return -1;
    }

    return( read( filedes, buffer, rec_size ) );
}
```

The `read_record()` function in this example assumes records are numbered starting with zero, and that `rec_size` contains the size of a record in the file, including any record-separator characters.

Classification:

`lseek()` is POSIX 1003.1; `lseek64()` is Large-file support.
Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chsize(), close(), creat(), dup(), dup2(), eof(), errno, execl(), execlp(), execle(), execvp(), execve(), fcntl(), fileno(), fstat(), isatty(), open(), read(), sopen(), stat(), tell(), umask(), write()
**lstat(), lstat64()**

Get information about a file or directory

**Synopsis:**

```c
#include <sys/stat.h>

int lstat( const char* path,
           struct stat* buf );

int lstat64( const char* path,
             struct stat64* buf );
```

**Arguments:**

- `path` The path of the file or directory that you want information about.
- `buf` A pointer to a buffer where the function can store the information.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

These functions obtain information about the file or directory referenced in `path`. This information is placed in the structure located at the address indicated by `buf`.

The results of the `lstat()` function are the same as the results of `stat()` when used on a file that isn’t a symbolic link. If the file is a symbolic link, `lstat()` returns information about the symbolic link, while `stat()` continues to resolve the pathname using the contents of the symbolic link, and returns information about the resulting file.
Returns:

0  Success.

-1  An error occurred (errno is set).

Errors:

See stat() for details.

Examples:

```c
/*
 * Iterate through a list of files, and report
 * for each if it is a symbolic link
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    int ecode = 0;
    int n;
    struct stat sbuf;

    for( n = 1; n < argc; ++n ) {
        if( lstat( argv[n], &sbuf ) == -1 ) {
            perror( argv[n] );
            ecode++;
        } else if( S_ISLNK( sbuf.st_mode ) ) {
            printf( "%s is a symbolic link\n", argv[n] );
        } else {
            printf( "%s is not a symbolic link\n", argv[n] );
        }
    }

    return( ecode );
}
```

Classification:

lstat() is POSIX 1003.1; lstat64() is Large-file support
lstat(), lstat64()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

erro, fstat(), readlink(), stat()
**Synopsis:**

```c
#include <stdlib.h>

char* ltoa( long value, 
            char* buffer, 
            int radix);

char* lltoa( long long value, 
             char* buffer, 
             int radix);
```

**Arguments:**

- **value**  
The value to convert into a string.
- **buffer**  
A buffer in which the function stores the string. The size of the buffer must be at least 33 bytes when converting values in base 2 (binary).
- **radix**  
The base to use when converting the number. This value must be in the range:

\[
2 \leq \text{radix} \leq 36
\]

If the value of `radix` is 10, and `value` is negative, then a minus sign is prepended to the result.

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ltoa()` and `lltoa()` functions convert the given long integer `value` into the equivalent string in base `radix` notation, storing the result in the character array pointed to by `buffer`. A NUL character is appended to the result.
Returns:

A pointer to the result.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

void print_value( long value )
{
    int base;
    char buffer[33];
    for( base = 2; base <= 16; base = base + 2 ) {
        printf( "%2d %s\n", base,
                ltoa( value, buffer, base ) );
    }
}

int main( void )
{
    print_value( 12765 );
    return EXIT_SUCCESS;
}
```

produces the output:

```
2 11000111011101
4 3013131
6 135033
8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

ltoa() is QNX 4; lltoa() is Unix

Safety

Cancellation point  No

continued…
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atoi()`, `atol()`, `itoa()`, `sscanf()`, `strtol()`, `strtoul()`, `ultoa()`, `utoa()`
**ltrunc()**

Truncate a file at a given position

**Synopsis:**

```c
#include <sys/types.h>
#include <unistd.h>

off_t ltrunc( int fildes,
              off_t offset,
              int whence );
```

**Arguments:**

- `fildes` The file descriptor of the file that you want to truncate.
- `offset` The relative offset from the file position determined by the `whence` argument.
- `whence` The position in the file. The possible values (defined in `<unistd.h>`) are:
  - `SEEK_CUR`: The new file position is computed relative to the current file position. The value of `offset` may be positive, negative or zero.
  - `SEEK_END`: The new file position is computed relative to the end of the file.
  - `SEEK_SET`: The new file position is computed relative to the start of the file. The value of `offset` must not be negative.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ltrunc()` function attempts to truncate the file at a specified position. The file, referenced by the open file descriptor `fildes`, must have been opened `O_WRONLY` or `O_RDWR`. The truncation point is
calculated using the value of offset as a relative offset from a file position determined by the value of the argument whence. The value of offset may be negative, although a negative truncation point (one before the beginning of the file) is an error.

The ltrunc() function ignores advisory locks that may have been set by fcntl().

The calculated truncation point, if within the existing bounds of the file, determines the new file size; all data after the truncation point no longer exists. If the truncation point is past the existing end of file, the file size isn’t changed. An error occurs if you attempt to truncate before the beginning of the file (that is, a negative truncation point).

The current seek position isn’t changed by this function under any circumstance, including the case where the current seek position is beyond the truncation point.

Returns:

Upon successful completion, this function returns the new file size. If a truncation point beyond the existing end of file was specified, the existing file size is returned, and the file size remains unchanged. Otherwise, ltrunc() returns a value of -1 and sets errno to indicate the error. The file size remains unchanged in the event of an error.

Errors:

EBADF The fildes argument isn’t a valid file descriptor, open for writing.

EINVAL The whence argument isn’t a proper value, or the resulting file size would be invalid.

ENOSYS An attempt was made to truncate a file of a type that doesn’t support truncation (for example, a file associated with the device manager).
ESPIPE

The fildes argument is associated with a pipe or FIFO.

Examples:

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

char buffer[1000];

int main( void )
{
    int fd, stat;
    fd = open( "test", O_CREAT | O_RDWR, 0666 );
    if( fd == -1 ) {
        fprintf( stderr, "Open error\n" );
        exit( -1 );
    }

    /* Create a 1000-byte file */
    write( fd, buffer, 1000 );

    /* Seek back to offset 500 and truncate the file */
    if( ltrunc( fd, 500, SEEK_SET ) == -1 ) {
        fprintf( stderr, "ltrunc error\n" );
        exit( -1 );
    }

    close( fd );
    fd = open( "test", O_CREAT | O_RDWR, 0666 );
    printf( "File size = %ld\n",
           lseek( fd, 0, SEEK_END ) );
    close( fd );

    return 0;
}
```

Classification:

QNX 4

Safety

Cancellation point  Yes

continued…
**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The `ltrunc()` function isn’t portable, and shouldn’t be used in new code. Use `ftruncate()` instead.

**See also:**

`errno`, `ftruncate()`, `lseek()`
C Library — M to O
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td><code>abort()</code> to <code>expm1f()</code></td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td><code>fabs()</code> to <code>hypotf()</code></td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to <code>lrint()</code></td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td><code>main()</code> to <code>outle32()</code></td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td><code>pathconf()</code> to <code>ruserok()</code></td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td><code>sbrk()</code> to <code>system()</code></td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td><code>tan()</code> to <code>ynf()</code></td>
</tr>
</tbody>
</table>
**main()**

Program entry function

**Synopsis:**

```c
int main( void );

int main( int argc,  
        const char *argv[] );

int main( int argc,  
        const char *argv[],  
        char *envp[] );
```

**Arguments:**

The arguments depend on which form of `main()` that you use.

- `argc` The number of entries in the `argv` array.
- `argv` An array of pointers to strings that contain the arguments to the program.
- `envp` An array of pointers to strings that define the environment for the program.

**Description:**

The `main()` function is supplied by the user and is where program execution begins. The command line to the program is broken into a sequence of tokens separated by blanks, and are passed to `main()` as an array of pointers to character strings in `argv`. The number of arguments found is passed in the parameter `argc`.

The `argv[0]` argument is a pointer to a character string containing the program name. The last element of the array pointed to by `argv` is NULL (`argv[argc]` is NULL). Arguments containing blanks can be passed to `main()` by enclosing them in quote characters (which are removed from that element in the `argv` vector). See your shell’s documentation for details.

The `envp` argument points to an array of pointers to character strings that are the environment strings for the current process. This value is identical to the `environ` variable, which is defined in the `<stdlib.h>` header file.
**Returns:**

A value back to the calling program (usually the operating system).

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    int i;
    for( i = 0; i < argc; ++i ) {
        printf( "argv[%d] = %s\n", i, argv[i] );
    }

    return EXIT_SUCCESS;
}
```

produces the output:

```
argv[0] = ./mypgm
argv[1] = hhhhh
argv[2] = another arg
```

when the program `mypgm` is run from the shell:

```
$ ./mypgm hhhhh "another arg"
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

abort(), atexit(), _argc, _argv, _auxv, close(), execl(), execle(),
exelpl(), execlepl(), execv(), execve(), execvp(), execvep(), _exit(),
exit(), getenv(), putenv(), sigaction(), signal(), spawn(), spawnl(),
spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnvp(), spawnv(), spawnve(),
spawnvp(), spawnvpe(), system(), wait(), waitpid()
mallinfo()

Get memory allocation information

Synopsis:

```c
#include <malloc.h>

struct mallinfo mallinfo ( void );
```

Library:

libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `mallinfo()` function returns memory-allocation information in the form of a `struct mallinfo`:

```c
struct mallinfo {
    int arena;    /* size of the arena */
    int ordblks;  /* number of big blocks in use */
    int smblks;   /* number of small blocks in use */
    int hblks;    /* number of header blocks in use */
    int hbbhd;    /* space in header block headers */
    int usmblds;  /* space in small blocks in use */
    int fsmblds;  /* memory in free small blocks */
    int fbodbks;  /* space in big blocks in use */
    int fodboks;  /* memory in free big blocks */
    int keepcost; /* penalty if M_KEEP is used */
};
```

Returns:

A `struct mallinfo`.

Classification:

QNX Neutrino
mallinfo()  © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

calloc(), free(), malloc(), realloc()
Synopsis:

```c
#include <stdlib.h>

void* malloc( size_t size );
```

Arguments:

- `size` The number of bytes to allocate.

Library:

- libc
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `malloc()` function allocates a buffer of `size` bytes.

If `size` is zero, `malloc()` returns a valid non-NULL pointer which is valid only to a corresponding `free()` call. You shouldn’t assume that a pointer returned by `malloc(0)` actually points to any valid memory.

This function allocates memory in blocks of `_ambksz` bytes (a global variable defined in `<stdlib.h>`).

Returns:

- A pointer to the start of the allocated memory, or NULL if an error occurred (`errno` is set).

Errors:

- ENOMEM Not enough memory.
- EOK No error.
Examples:

```c
#include <stdlib.h>

int main( void )
{
    char* buffer;

    buffer = (char*) malloc( 80 );
    if( buffer != NULL ) {
        /* do something with the buffer */
        ...

        free( buffer );
    }

    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

In QNX 4, nothing is allocated when you `malloc()` 0 bytes. Be careful if your code is ported between QNX 4 and QNX Neutrino.

See also:

`calloc()`, `free()`, `realloc()`, `sbrk()`

The Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino *Programmer’s Guide*. 
Synopsis:

```c
#include <malloc.h>

int mallopt( int cmd,
              int value );
```

Arguments:

- `cmd` Options used to enable additional checks in the library.
  - MALLOCS\_CKACCESS
  - MALLOCS\_FILLAREA
  - MALLOCS\_CKCHAIN
  - MALLOCS\_VERIFY
  - MALLOCS\_FATAL
  - MALLOCS\_WARN
  
  See the Description section for more details.

- `value` A value corresponding to the command; see below:

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mallopt()` function controls the extra checking of memory allocation.

Options used to enable additional checks in the library include:

- **MALLOCS\_CKACCESS**
  
  Turn on (or off) boundary checking for memory and string operations.
  
  Environment variable: `MALLOCS\_CKACCESS`.

The `value` argument can be:
mallopt()

- zero to disable the checking
- nonzero to enable it.

MALLOCFILLAREA

Turn on (or off) fill-area boundary checking.
Environment variable: MALLOCFILLAREA.

The value argument can be:
- zero to disable the checking
- nonzero to enable it.

MALLOCKCHAIN

Enable (or disable) full-chain checking. For each of the above options, an integer argument value of one indicates that the given type of checking should be enabled from that point onward.
Environment variable: MALLOCKCHAIN.

The value argument can be:
- zero to disable the checking
- nonzero to enable it.

MALLOCFVERIFY

Perform a chain check. If an error is found, perform error handling. The value argument is currently ignored; pass 1 for it.

MALLOCFATAL

Specify the malloc fatal handler.
Environment variable: MALLOCFATAL.

Use one of the following for the value arguments:
mallopt()

<table>
<thead>
<tr>
<th>Symbolic name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_HANDLE_IGNORE 0</td>
<td>0</td>
<td>Cause the program to dump a core file.</td>
</tr>
<tr>
<td>M_HANDLE_ABORT 1</td>
<td>1</td>
<td>Terminate execution with a call to abort()</td>
</tr>
<tr>
<td>M_HANDLE_ABORT 2</td>
<td>2</td>
<td>Exit immediately</td>
</tr>
<tr>
<td>M_HANDLE_CORE 3</td>
<td>3</td>
<td>Cause the program to dump a core file</td>
</tr>
<tr>
<td>M_HANDLE_SIGNAL 4</td>
<td>4</td>
<td>Stop the program when this error occurs</td>
</tr>
</tbody>
</table>

MALLOCP_WARN

Specify the malloc warning handler. The values are similar to MALLOCP_FATAL; see above.

Environment variable: MALLOCP_WARN.

For details, see “Controlling the level of checking” in the Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino Programmer’s Guide.

See the Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino Programmer’s Guide.

Returns:

0 on success, or -1 if an error occurs (errno is set).

Classification:

QNX Neutrino
mallopt()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

calloc(), free, mallinfo(), malloc, realloc()

Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino Programmer’s Guide
max()

Return the greater of two numbers

Synopsis:

```c
#include <stdlib.h>

#define max(a, b) ...
```

Arguments:

- `a`, `b` The numbers that you want to get the greater of.

Description:

The `max()` function returns the greater of two values.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int a;

    a = max( 1, 10 );
    printf( "The value is: %d\n", a );
    return EXIT_SUCCESS;
}
```

Classification:

QNX 4

Safety

- Cancellation point: No
- Interrupt handler: Yes

continued…
max()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

min()
Synopsis:

```
#include <stdlib.h>

int mblen( const char * s,
           size_t n );
```

Arguments:

- `s` NULL (see below), or a pointer to a multibyte character.
- `n` The maximum number of bytes that you want to count.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mblen()` function counts the number of bytes in the multibyte character pointed to by `s`, to a maximum of `n` bytes.

The `mbrlen()` function is a restartable version of `mblen()`.

Returns:

- If `s` is NULL, `mblen()` determines whether or not the character encoding is state-dependent:
  - `0` The `mblen()` function uses locale-specific multibyte character encoding that’s not state-dependent.
  - `≠ 0` Character is state-dependent.

- If `s` isn’t NULL:
  - `0` `s` points to the null character.
  - `-1` The next `n` bytes don’t form a valid multibyte character.
  - `> 0` The number of bytes that comprise the multibyte character (if the next `n` or fewer bytes form a valid multibyte character).
**mblen()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int len;
    char *mbs = "string";

    printf( "Character encodings do " );
    if( !mblen( NULL, 0 ) ) {
        printf( "not " );
    }
    printf( "have state-dependent \nencoding.\n" );

    len = mblen( "string", 6 );
    if( len == -1 ) {
        mbs[len] = '\0';
        printf( "Multibyte char 's'(%d)\n", mbs, len );
    }

    return EXIT_SUCCESS;
}
```

This produces the output:

Character encodings do not have state-dependent encoding.
Multibyte char 's'(1)

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**Synopsis:**

```c
#include <wchar.h>

size_t mbrlen( const char * s,
                size_t n,
                mbstate_t * ps);
```

**Arguments:**

- **s** A pointer to a multibyte character.
- **n** The maximum number of bytes that you want to count.
- **ps** An internal pointer that lets `mbrlen()` be a restartable version of `mblen()`; if `ps` is NULL, `mbrlen()` uses its own internal variable.

You can call `mbsinit()` to determine the status of this variable.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mbrlen()` function counts the bytes in the multibyte character pointed to by `s`, to a maximum of `n` bytes.

**Returns:**

- `(size_t)-2` The resulting conversion state indicates an incomplete multibyte character after all `n` characters were converted.
- `(size_t)-1` The function detected an encoding error before completing the next multibyte character, in which case the function `errno` to EILSEQ and leaves the resulting conversion state undefined.
mbrlen()

0 The next completed character is a null character, in which case the resulting conversion state is the initial conversion state.

x The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that x bytes have been converted.

Errors:

EILSEQ Invalid character sequence.

EINVAL The ps argument points to an invalid object.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Convert a multibyte character into a wide character (restartable)

Synopsis:
```
#include <wchar.h>

size_t mbrtowc( wchar_t * pwc, const char * s, size_t n, mbstate_t * ps );
```

Arguments:
- **pwc**: A pointer to a wchar_t object where the function can store the wide character.
- **s**: A pointer to the multibyte character that you want to convert.
- **n**: The maximum number of bytes in the multibyte character to convert.
- **ps**: An internal pointer that lets mbrtowc() be a restartable version of mbtowc(); if ps is NULL, mbrtowc() uses its own internal variable.
  You can call mbsinit() to determine the status of this variable.

Library:
```
libc
```
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The mbrtowc() function converts single multibyte characters pointed to by s into wide characters pointed to by pwc, to a maximum of n bytes (not characters).
This function is affected by LC_TYPE.
**mbrtowc()**

**Returns:**

- `(size_t)-2` After converting all *n* characters, the resulting conversion state indicates an incomplete multibyte character.
- `(size_t)-1` The function detected an encoding error before completing the next multibyte character; the function sets *errno* to EILSEQ and leaves the resulting conversion state undefined.
- `0` The next completed character is a null character; the resulting conversion state is the same as the initial one.
- `x` The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that *x* bytes have been converted.

**Errors:**

- EILSEQ Invalid character sequence.
- EINV AL The *ps* argument points to an invalid object.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

*errno*

“Multibyte character functions,” “Stream I/O functions,” and “Wide-character functions” in the summary of functions chapter.
Determine the status of the conversion object used for restartable mb*() functions

Synopsis:

```c
#include <wchar.h>

int mbsinit( const mbstate_t * ps );
```

Arguments:

- `ps` A pointer to the conversion object that you want to test.

Library:

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The following functions use an object of type `mbstate_t` so that they can be restarted:

- `mbrlen()`
- `mbrtowc()`
- `mbsrtowcs()`
- `mbstowcs()`
- `wcsrtombs()`
- `wcrtomb()`

The `mbsinit()` function determines whether or not the `mbstate_t` object pointed to by `ps` describes an initial conversion state.
If the object doesn’t describe an initial conversion state, it isn’t safe
for you to use it in one of the above functions, other than the one
you’ve already used it in.

Returns:

A nonzero value if \( ps \) is NULL or \(^{\ast}ps \) describes an initial conversion
state; otherwise zero.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

\( mbrlen() \), \( mbtowc() \), \( mbsrtowcs() \), \( mbstowcs() \), \( wcsrtombs() \),
\( wcrtomb() \)

“Multibyte character functions” and “Wide-character functions” in
the summary of functions chapter.
### mbsrtowcs()

**Convert a multibyte-character string into a wide-character string (restartable)**

### Synopsis:

```c
#include <wchar.h>

size_t mbsrtowcs( wchar_t * dst,
                  const char ** src,
                  size_t n,
                  mbstate_t * ps );
```

### Arguments:

- **dst**: A pointer to a buffer where the function can store the wide-character string.
- **src**: The string of multibyte characters that you want to convert.
- **n**: The maximum number of bytes that you want to convert.
- **ps**: An internal pointer that lets `mbsrtowcs()` be a restartable version of `mbstowcs()`; if `ps` is NULL, `mbsrtowcs()` uses its own internal variable. You can call `mbsinit()` to determine the status of this variable.

### Library:

- **libc**
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `mbsrtowcs()` function converts a string of multibyte characters pointed to by `src` into the corresponding wide characters pointed to by `dst`, to a maximum of `n` bytes, including the terminating NULL character.

The function converts each character as if by a call to `mbtowc()` and stops early if:

- A sequence of bytes doesn’t conform to a valid character
mbsrtowcs() © 2007, QNX Software Systems GmbH & Co. KG.

Or:
- Converting the next character would exceed the limit of \( n \) total bytes.

This function is affected by LC_TYPE.

Returns:

\[
\begin{align*}
(size_t)-1 & \quad \text{Failure; invalid wide-character code.} \\
x & \quad \text{Success; the number of total bytes successfully converted, not including the terminating NULL byte.}
\end{align*}
\]

Errors:

- EILSEQ Invalid character sequence.
- EINVAL The \( ps \) argument points to an invalid object.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

errno

“Multibyte character functions” and “Wide-character functions” in the summary of functions chapter.
mbstowcs()

Convert a multibyte-character string into a wide-character string

Synopsis:

```c
#include <stdlib.h>

size_t mbstowcs( wchar_t * pwcs,
    const char * s,
    size_t n );
```

Arguments:

- `pwcs` A pointer to a buffer where the function can store the
  wide-character string.
- `s` The string of multibyte characters that you want to convert.
- `n` The maximum number of bytes that you want to convert.

Library:

```
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `mbstowcs()` function converts a sequence of multibyte characters
pointed to by `s` into their corresponding wide-character codes pointed
to by `pwcs`, to a maximum of `n` bytes. It doesn’t convert any multibyte
characters beyond a NULL character.

This function is affected by LC_TYPE.

The `mbsrtowcs()` function is a restartable version of `mbstowcs()`.

Returns:

The number of array elements modified, not including the terminating
zero code, if present, or `(size_t)-1` if an invalid multibyte
character was encountered.
mbstowcs()

Errors:

EILSEQ         Invalid character sequence.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *wc = "string";
    wchar_t wbuffer[50];
    int i, len;

    len = mbstowcs( wbuffer, wc, 50 );
    if( len != -1 ) {
        wbuffer[len] = '\0';
        printf( "%s(%d)
", wc, len );
        for( i = 0; i < len; i++ ) {
            printf( "/%4.4x", wbuffer[i] );
        }
        printf( "\n" );
    }

    printf( "\n" );

    return EXIT_SUCCESS;
}
```

This produces the output:

```
string(6)
/0073/0074/0072/0069/0067
```

Classification:

ANSI, POSIX 1003.1

Safety

Cancellation point  No
Interrupt handler   No

continued...
© 2007, QNX Software Systems GmbH & Co. KG.

**mbstowcs()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*errno*

“Multibyte character functions” and “Wide-character functions” in the summary of functions chapter.
**mbtowc()**

Convert a multibyte character into a wide character

**Synopsis:**

```c
#include <stdlib.h>

int mbtowc( wchar_t * pwc,
            const char * s,
            size_t n );
```

**Arguments:**

- **pwc** A pointer to a `wchar_t` object where the function can store the wide character.
- **s** NULL (see below), or a pointer to the multibyte character that you want to convert.
- **n** The maximum number of bytes in the multibyte character to convert.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mbtowc()` function converts a single multibyte character pointed to by `s` into a wide-character code pointed to by `pwc`, to a maximum of `n` bytes. The function stops early if it encounters the NULL character.

This function is affected by LC_TYPE.

The `mbtowc()` function is a restartable version of `mbtowc()`.

**Returns:**

- If `s` is NULL:
  
  0 The `mbtowc()` function uses UTF-8 multibyte character encoding that’s not state-dependent.
≠ 0  Everything else.

- If s isn’t NULL:
  0  The s argument points to the NUL character.
  > 0  The number of bytes that comprise the multibyte character, to a maximum of MB_CUR_MAX (if the next n or fewer bytes form a valid multibyte character).
  -1  The next n bytes don’t form a valid multibyte character; errno is set.

Errors:

EILSEQ  Invalid character sequence.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *wc = "string";
    wchar_t wbuffer[10];
    int i, len;

    printf( "State-dependent encoding? " );
    if( mbtowc( wbuffer, NULL, 0 ) ) {
        printf( "Yes\n" );
    } else {
        printf( "No\n" );
    }

    len = mbtowc( wbuffer, wc, 2 );
    wbuffer[len] = ‘\0’;
    printf( "%s(%d)\n", wc, len );

    for( i = 0; i < len; i++ ) {
        printf( "/%4.4x", wbuffer[i] );
    }

    printf( "\n*" );
    return EXIT_SUCCESS;
}
```
This produces the output:

```
State-dependent encoding? No
string(1)
/0073
```

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*errno*

“Multibyte character functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```c
#include <malloc.h>

int mcheck(
    void (* abort_fn) (enum mcheck_status status));
```

Arguments:

- `abort_fn` A pointer to the callback function to invoke when an inconsistency in the memory-allocation routines is found, or NULL if you want to use the default callback routine.

The argument to the callback routine is one of the values of the `mcheck_status` enumeration described in the documentation for `mprobe()`.

The default abort callback prints a message to `stderr` and aborts the application.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mcheck()` function enables consistency checks within the memory allocation routines. When enabled, consistency checks are periodically performed on allocated memory blocks as blocks are allocated or freed. If an inconsistency is found, the `abort` callback is called with the status identifying the type of inconsistency found.

Consistency checking isn’t performed on blocks that you allocated before calling `mcheck()`.

The level of checking provided depends on which version of the allocator you’ve linked the application with:
- C library — minimal consistency checking.
- Nondebug version of the malloc library — a slightly greater level of consistency checking.
- Debug version of the malloc library — extensive consistency checking, with tuning available through the use of the mallopt() function.

You can call mcheck() only once in a program.

Returns:

-1 Checking is already enabled.
0 Checking wasn’t already enabled.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

mallopt(), mprobe()

Heap Analysis in the Programmer’s Guide
Synopsis:

```c
#include <sys/mman.h>

int mem_offset( const void *addr, int fd, size_t length, off_t *offset, size_t *contig_len);

int mem_offset64( const void *addr, int fd, size_t length, off64_t *offset, size_t *contig_len);
```

Arguments:

- `addr` The address of the memory block whose offset and contiguous length you want to get.
- `fd` The file descriptor that identifies the typed memory object. This must be the descriptor that you used (in a call to `mmap()`) to establish the mapping that contains `addr`.
- `length` The length of the block of memory that you want the offset for.
- `offset` A pointer to a location where the function can store the offset of the memory block.
- `contig_len` A pointer to a location where the function can store either `length` or the length of the largest contiguous block of typed memory that's currently mapped to the calling process starting at `addr`, whichever is smaller.
**mem_offset(), mem_offset64()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mem_offset()` and `mem_offset64()` functions set the variable pointed to by `offset` to the offset (or location), within a typed memory object, of the memory block currently mapped at `addr`.

If you use the `offset` and `contig_len` values obtained from calling `mem_offset()` in a call to `mmap()` with a file descriptor that refers to the same memory pool as `fd` (either through the same port or through a different port), the memory region that’s mapped must be exactly the same region that was mapped at `addr` in the address space of the process that called `mem_offset()`.

For best performance results, it is preferable to cache the result of `mem_offset()`, rather than repeatedly call the function for a given virtual address.

**QNX extension**

If you specify `fd` as NOFD, `offset` is the offset into `/dev/mem` of `addr` (i.e. its physical address). If the memory object specified by `fd` isn’t a typed memory object, or specified as NOFD, the call fails.
mem_offset(),
mem_offset64()

If the physical address is not a valid off_t value, mem_offset() will fail with errno set to E2BIG. This is typically the case with many ARM systems, and you should use mem_offset64() to get the physical address.

For the NOFD case, mem_offset():

- Cause the initial copying or zero-filling of the MAP_PRIVATE or MAP_ANON pages.
- Return -1 (EACCES) for MAP_LAZY pages that are not memory resident yet.

Returns:

- 0   Success.
- -1  An error occurred (errno is set).

Errors:

- EACCES The process hasn’t mapped memory at the given address addr.
- EBADF Invalid open file descriptor fildes.
- EINVAL The file descriptor fildes doesn’t correspond to the memory object mapped at addr.
- ENODEV The file descriptor fildes isn’t connected to a memory object supported by this function.
- ENOSYS The mem_offset() function isn’t supported by this implementation.
Examples:

In the example below, \textit{offset} contains the physical address of the memory mapped at the given address \textit{addr}.

\begin{verbatim}
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/mman.h>

paddr_t mphys(void *addr) {
    off64_t offset;
    if(mem_offset64(addr, NOFD, 1, &offset, 0) == -1) {
        return -1;
    }
    return offset;
}
\end{verbatim}

Classification:

\textit{mem_offset()} is QNX Neutrino; \textit{mem_offset64()} is Large-file support

\begin{tabular}{ll}
\textbf{Safety} & \\
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}

See also:

\textit{mmap()}, \textit{posix_mem_offset()}, \textit{posix_mem_offset64()}

---

\textit{mem_offset()}, \textit{mem_offset64()} © 2007, QNX Software Systems GmbH & Co. KG.
memalign()
Allocate aligned memory

Synopsis:

```c
#include <malloc.h>

void *memalign( size_t alignment,
                size_t size );
```

Arguments:

- `alignment` The alignment that you want to use for the memory. This must be a multiple of `size( void *)`.
- `size` The amount of memory you want to allocate, in bytes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memalign()` function allocates `size` bytes aligned on a boundary specified by `alignment`.

Returns:

A pointer to the allocated block, or NULL if an error occurred (`errno` is set).

Errors:

- `EINVAL` The value of `alignment` isn’t a multiple of `size( void *)`.
- `ENOMEM` There’s insufficient memory available with the requested alignment.
memalign()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

free(), malloc(), posix_memalign()
memccpy()
Copy bytes between buffers until a given byte is found

Synopsis:

```c
#include <string.h>

void* memccpy( void* dest,
    const void* src,
    int c,
    size_t cnt );
```

Arguments:
- `dest` A pointer to where you want the function to copy the data.
- `src` A pointer to the buffer that you want to copy data from.
- `c` The value that you want to stop copying at.
- `cnt` The maximum number of bytes to copy.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memccpy()` function copies bytes from `src` to `dest`, up to and including the first occurrence of the character `c`, or until `cnt` bytes have been copied, whichever comes first.

Returns:

A pointer to the byte in `dest` following the character `c`, if one is found and copied; otherwise, NULL.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```
char* msg = "This is the string: not copied";

int main( void )
{
    char buffer[80];

    memset( buffer, '\0', 80 );
    memccpy( buffer, msg, ':', 80 );

    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}

produces the output:

This is the string:

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memchr()`, `memcmp()`, `memcpy()`, `memcmp()`, `memmove()`, `memset()`
Synopsis:

```
#include <string.h>

void* memchr(void* buf, int ch, size_t length);
```

Arguments:

- `buf` The buffer that you want to search.
- `ch` The character that you’re looking for.
- `length` The number of bytes to search in the buffer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memchr()` function locates the first occurrence of `ch` (converted to an `unsigned char`) in the first `length` bytes of the buffer pointed to by `buf`.

Returns:

A pointer to the located character, or NULL if `ch` couldn’t be found.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
    char* where;
```
strcpy( buffer, "video x-rays" );

where = (char *) memchr( buffer, 'x', 6 );
if( where == NULL ) {
   printf( "'x' not found\n" );
} else {
   printf( "%s\n", where );
}

where = (char *) memchr( buffer, 'r', 9 );
if( where == NULL ) {
   printf( "'r' not found\n" );
} else {
   printf( "%s\n", where );
}

return EXIT_SUCCESS;
}

produces the output:

'x' not found
rays

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memcpy(), memcmp(), memcpy(),memcmpp(), memmove(), memset(),
strchr(), strrchr(), wmemchr(), wmemcmp(), wmemcpy(),
wmemmove(), wmemset()
Synopsis:

```c
#include <string.h>

int memcmp( const void* s1, const void* s2, size_t length );
```

Arguments:

- `s1`, `s2` Pointers to the buffers that you want to compare.
- `length` The number of bytes that you want to compare.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memcmp()` function compares `length` bytes of the buffer pointed to by `s1` to the buffer pointed to by `s2`.

Returns:

- `< 0` The object pointed to by `s1` is less than the object pointed to by `s2`.
- `0` The object pointed to by `s1` is equal to the object pointed to by `s2`.
- `> 0` The object pointed to by `s1` is greater than the object pointed to by `s2`.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```
int main( void )
{
    char buffer[80];
    int retval;

    strcpy( buffer, "World" );

    retval = memcmp( buffer, "hello", 5 );
    if( retval < 0 ) {
        printf( "Less than\n" );
    } else if( retval == 0 ) {
        printf( "Equal to\n" );
    } else {
        printf( "Greater than\n" );
    }

    return EXIT_SUCCESS;
}

produces the output:

Less than

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memccpy()`, `memchr()`, `memcpy()`, `memicmp()`, `memmove()`, `memset()`
memcpy()

Copy bytes from one buffer to another

Synopsis:

```c
#include <string.h>

void* memcpy( void* dst,
  const void* src,
  size_t length );
```

Arguments:

- `dest` A pointer to where you want the function to copy the data.
- `src` A pointer to the buffer that you want to copy data from.
- `length` The number of bytes to copy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memcpy()` function copies `length` bytes from the buffer pointed to by `src` into the buffer pointed to by `dst`.

Copying overlapping buffers isn’t guaranteed to work; use `memmove()` to copy buffers that overlap.

Returns:

A pointer to the destination buffer (that is, the value of `dst`).

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
```


```c
{
    char buffer[80];
    memcpy( buffer, "Hello", 5 );
    buffer[5] = '\0';
    printf( "%s\n", buffer );
    return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

---

### Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

---

**See also:**

`memcpy()`, `memchr()`, `memcmp()`, `memcmp()`, `memmove()`, `memset()`
memcpyv()

Copy a given number of structures

Synopsis:
```c
#include <string.h>

size_t memcpyv( const struct iovec *dst,
                 int dparts,
                 int doff,
                 const struct iovec *src,
                 int sparts,
                 int soff );
```

Arguments:
- `dst` — An array of `iovec` structures that you want to copy the data to.
- `dparts` — The number of entries in the `dst` array.
- `doff` — The offset into the `dst` array at which to start copying.
- `src` — An array of `iovec` structures that you want to copy the data from.
- `sparts` — The number of entries in the `src` array.
- `soff` — The offset into the `src` array at which to start copying.

Library:
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The function `memcpyv()` copies data pointed to by the `src` I/O vector, starting at offset `soff`, to `dst` structures, starting at offset `doff`. The number of I/O vector parts copied is specified in `sparts` and `dparts`. 
memcpyv()

Returns:

The number of bytes copied.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    const struct iovec *dest, *source;
    int dparts, doffset, sparts, soffset;
    size_t nbytes;

    nbytes = memcpyv ( dest, dparts, doffset,
                       source, sparts, soffset );
    printf ( "The number of bytes copied is %d. \n", nbytes );
    return EXIT_SUCCESS;
}
```

Classification:

QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memccpy()`, `memcpy()`
Synopsis:

```c
#include <string.h>

int memicmp( const void* s1,
             const void* s2,
             size_t length );
```

Arguments:

- `s1`, `s2` Pointers to the buffers that you want to compare.
- `length` The number of bytes that you want to compare.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memicmp()` function compares (case insensitive) `length` bytes of the buffer pointed to by `s1` with those of the buffer pointed to by `s2`.

Returns:

- `0` The object pointed to by `s1` is the same as the object pointed to by `s2`.
- Less than `0` The object pointed to by `s1` is less than the object pointed to by `s2`.
- Greater than `0` The object pointed to by `s1` is greater than the object pointed to by `s2`.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```
int main( void )
{
    char buffer[80];
    int retval;

    strncpy( buffer, "World" );

    retval = memcmp( buffer, "hello", 5 );
    if( retval < 0 ) {
        printf( "Less than\n" );
    } else if( retval == 0 ) {
        printf( "Equal\n" );
    } else {
        printf( "Greater than\n" );
    }

    return EXIT_SUCCESS;
}

produces the output:

Less than

Classification:

QNX 4

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memccpy()`, `memchr()`, `memcmp()`, `memcpy()`, `memmove()`, `memset()`
memmove()
Copy bytes from one buffer to another, handling overlapping memory correctly

Synopsis:

```
#include <string.h>

void* memmove( void* dst, const void* src, size_t length);
```

Arguments:

- `dest` A pointer to where you want the function to copy the data.
- `src` A pointer to the buffer that you want to copy data from.
- `length` The number of bytes to copy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memmove()` function copies `length` bytes from the buffer pointed to by `src` to the buffer pointed to by `dst`. Copying of overlapping regions is handled safely. Use `memcpy()` for greater speed when copying buffers that don’t overlap.

Returns:

A pointer to the destination buffer (that is, the value of `dst`).

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
```
```c
strcpy( buffer, "World" );
memmove( buffer+1, buffer, 79 );
printf ("%s \n", buffer);

return EXIT_SUCCESS;
```

produces the output:

```
WWorld
```

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memcpy()`, `memchr()`, `memcmp()`, `memcpys()`, `memcpy()`, `memicmp()`, `memset()`, `wmemmove()`
Synopsis:

```c
#include <string.h>

void* memset( void* dst,
              int c,
              size_t length );
```

Arguments:
- `dst` A pointer to the memory that you want to set.
- `c` The value that you want to store in each byte.
- `length` The number of bytes to set.

Library:

`libc`
Use the `-1 libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memset()` function fills `length` bytes starting at `dst` with the value `c`.

Returns:

A pointer to the destination buffer (that is, the value of `dst`).

Examples:

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    char buffer[80];
    memset( buffer, '=', 80 );
    buffer[79] = '\0';
}
```
memset()  © 2007, QNX Software Systems GmbH & Co. KG.

```c
puts( buffer );
return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memccpy()`, `memchr()`, `memcmp()`, `memcpy()`, `memcmp()`, `memmove()`
message_attach()

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int message_attach( dispatch_t * dpp,
                    message_attr_t * attr,
                    int low,
                    int high,
                    int (* func) (message_context_t * ctp,
                                  int code,
                                  unsigned flags,
                                  void * handle ),
                    void * handle );
```

Arguments:

- **dpp** The dispatch handle, as returned by `dispatch_create()`.
- **attr** A pointer to a `message_attr_t` structure that lets you specify additional requirements for the message; see “`message_attr_t` structure,” below.
- **low, high** The range of messages that you’re interested in.
- **func** The function that you want to call when a message in the given range is received; see “Handler function,” below.
- **handle** An arbitrary handle that you want to associate with data for the defined message range. This handle is passed to `func`.

Library:

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The message_attach() function attaches a handler to the message range defined by the message type [low, high] (i.e. an inclusive message range) for dispatch handle dpp.

It’s considered a programming error to attach overlapping message or pulse ranges. Message types should be greater than _IO_MAX (defined in <sys/iomsg.h>.

When a message with a type in that range is received, dispatch_handler() calls the user-supplied function func. You can also use the same function with pulse_attach(). By examining ctp->rcvid, func can determine whether a pulse or message was received.

This function is responsible for doing any specific work needed to handle the message pointed to by ctp->msg. The handle passed to the function is the handle initially passed to message_attach().

message_attr_t structure

The attr argument is a pointer to a message_attr_t structure:

typedef struct _message_attr {
  unsigned flags;
  unsigned nparts_max;
  unsigned msg_max_size;
} message_attr_t;

You can use this structure to specify:

- the maximum message size to be received (the context allocated must be at least big enough to contain a message of that size)

- the maximum number of iovs to reserve in the message_context_t structure (attr->nparts_max)

- various flags:
  Currently, the following attr->flags are defined:

  MSG_FLAG_CROSS_ENDIAN
  Allow the server to receive messages from clients on machines with different native endian formats.
MSG_FLAG_DEFAULT_FUNC

Call this function if no other match is found, in this case, low and high are ignored. This overrides the default behavior of dispatch_handler() which is to return MsgError() (ENOSYS) to the sender when an unknown message is received.

Handler function

The user-supplied function func is called when a message in the defined range is received. This function is passed the message context ctp, in which the message was received, the message type, and the handle (the one passed to message_attach()). Currently, the argument flags is reserved. Your function should return 0; other return values are reserved.

Here’s a brief description of the context pointer fields:

ctp->rcvid The receive ID of the message.
ctp->msg A pointer to the message.
ctp->info Data from a _msg_info structure.

Returns:

Zero on success, or -1 on failure (errno is set).

Errors:

EINVAL The message code is out of range.
ENOMEM Insufficient memory to attach message type.

Examples:

In this example, we create a resource manager where we attach to a private message range and attach a pulse, which is then used as a timer event:
```c
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#define THREAD_POOL_PARAM_T dispatch_context_t
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t connect_func;
static resmgr_io_funcs_t io_func;
static iofunc_attr_t attr;

int timer_tick(message_context_t *ctp, int code, unsigned flags, void *handle) {
    union sigval value = ctp->msg->pulse.value;
    /* Do some useful work on every timer firing... */
    printf("received timer event, value %d\n", value.sival_int);
    return 0;
}

int message_handler(message_context_t *ctp, int code, unsigned flags, void *handle) {
    printf("received private message, type %d\n", code);
    return 0;
}

int main(int argc, char **argv) {
    thread_pool_attr_t pool_attr;
    thread_pool_t *tpp;
    dispatch_t *dpp;
    resmgr_attr_t resmgr_attr;
    int id;
    int timer_id;
    struct sigevent event;
    struct _itimer itime;

    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr,
            "%s: Unable to allocate dispatch handle.\n", argv[0]);
        return EXIT_FAILURE;
    }
    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;

```
/* We are doing resmgr and pulse-type attaches */
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.unblock_func = dispatch_unblock;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

if((tpp = thread_pool_create(&pool_attr, 
    POOL_FLAG_EXIT_SELF)) == NULL) {
    fprintf(stderr, 
        "%s: Unable to initialize thread pool.\n", 
        argv[0]);
    return EXIT_FAILURE;
}

iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func,
    _RESMGR_IO_NFUNCS, &io_func);
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

if((id = resmgr_attach(dpp, &resmgr_attr, 
    "/dev/mynull", _FTYPE_ANY, 0, 
    &connect_func, &io_func, &attr)) == -1) {
    fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
    return EXIT_FAILURE;
}

/*
We want to handle our own private messages, of type 
0x5000 to 0x5fff
*/
if(message_attach(dpp, NULL, 0x5000, 0x5fff, 
    &message_handler, NULL) == -1) {
    fprintf(stderr, "Unable to attach to private message range.\n");
    return EXIT_FAILURE;
}

/* Initialize an event structure, and attach a pulse to it */
if((event.sigev_code = pulse_attach(dpp, 
    MSG_FLAGALLOC_PULSE, 0, 
    &timer_tick, NULL)) == -1) {
    fprintf(stderr, "Unable to attach timer pulse.\n");
message_attach()

return EXIT_FAILURE;

/* Connect to our channel */
if((event.sigev_coid = message_connect(dpp,
    MSG_FLAG_SIDE_CHANNEL)) == -1) {
    fprintf(stderr, "Unable to attach to channel.\n");
    return EXIT_FAILURE;
}

event.sigev_notify = SIGEV_PULSE;
event.sigev_priority = -1;
/*
   We could create several timers and use different
   sigev values for each
*/
event.sigev_value.sival_int = 0;

if((timer_id = TimerCreate(CLOCK_REALTIME, &event)) == -1) {;
    fprintf(stderr,
        "Unable to attach channel and connection.\n");
    return EXIT_FAILURE;
}

/* And now setup our timer to fire every second */
itime.nsec = 1000000000;
itime.interval_nsec = 1000000000;
TimerSettime(timer_id, 0, &itime, NULL);

/* Never returns */
thread_pool_start(tpp);
return EXIT_SUCCESS;

For more examples using the dispatch interface, see
dispatch_create(), resmgr_attach(), and thread_pool_create().

Classification:
QNX Neutrino

Safety
Cancellation point    Yes
Interrupt handler    No

continued...
**message_attach()**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

dispatch_block(), dispatch_create(), dispatch_handler(),
dispatch_unblock(), message_connect(), message_detach(),
_msg_info, pulse_attach()

“Components of a Resource Manager” section of the Writing a Resource Manager chapter in the *Programmer’s Guide*. 
message_connect()

Create a connection to a channel

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int message_connect( dispatch_t * dpp,
                   int flags );
```

Arguments:

- `dpp` The dispatch handle, as returned by `dispatch_create()`.
- `flags` Currently, the following flag is defined in `<sys/dispatch.h>`:
  - `MSG_FLAG_SIDECHELANNEL` — request the connection ID be returned from a different space. This ID will be greater than any valid file descriptor. Once created there’s no difference in the use of the messaging primitives on these IDs.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `message_connect()` function creates a connection to the channel used by dispatch handle `dpp`. This function calls the `ConnectAttach()` kernel call. To detach the connection ID, you can call `ConnectDetach()`.
The `message_connect()` function works only when the dispatch blocking type is receive, i.e. attaches were done for resmgr, message, or select "type" events. If no attaches were done yet, the `message_connect()` call fails, since dispatch can’t determine if receive or sigwait blocking will be used.

**Returns:**

A connection ID used by the message primitives, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EAGAIN** All kernel connection objects are in use.
- **EINVAL** Dispatch `dpp` doesn’t have a channel.

**Examples:**

```c
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int flags, coid, id;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate dispatch context.
", argv[0] );
        return EXIT_FAILURE;
    }

    id = resmgr_attach ( ... );

    if ( (coid = message_connect ( dpp, flags )) == -1 ) {
        fprintf ( stderr, "Failed to create connection to channel used by dispatch.\n" );
        return 1;
    }

    ...
}
```

September 10, 2007
message_connect()  

} /* else connection to channel used by dispatch is created */

:

For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

Dispatch dpp must block on messages.

See also:

ConnectAttach(), message_attach()

“Components of a Resource Manager” section of the Writing a Resource Manager chapter in the Programmer’s Guide.
message_detach()

Detach a message range

Synopsis:
```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int message_detach( dispatch_t * dpp,
        int    low,
        int    high,
        int    flags);
```

Arguments:
- **dpp**: The dispatch handle, as returned by `dispatch_create()`.
- **low, high**: The range of messages that you want to detach the handler from. This range must be the same one that you passed to `message_attach()`.
- **flags**: Reserved.

Library:
- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `message_detach()` function detaches the message type `[low, high]`, for dispatch handle `dpp`, that was attached with `message_attach()`.

Returns:
Zero on success. If an error occurs, -1 or the following error constant:

- **EINVAL**: The range `[low, high]` doesn’t match the range that you attached with `message_attach()`.
Examples:

```c
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int my_func(...)
{
...
}

int main( int argc, char **argv )
{
    dispatch_t *dpp;
    int lo=0x2000, hi=0x2fff, flags=0;

    if( ( dpp = dispatch_create() ) == NULL )
    {
        fprintf( stderr,
            "%s: Unable to allocate dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
    ...

    if( message_attach( dpp, NULL, lo, hi,
            &my_func, NULL) == -1 )
    {
        fprintf( stderr,
            "%s: Failed to attach message range.\n", argv[0] );
        return 1;
    }
    ...

    if ( message_detach ( dpp, lo, hi, flags ) == -1 )
    {
        fprintf ( stderr,
            "Failed to detach message range from %d to %d.\n", lo, hi );
        return 1;
    }
    /* else message was detached */
    ...
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.
message_detach()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

message_attach()

“Components of a Resource Manager” section of the Writing a Resource Manager chapter in the Programmer’s Guide.
**min()**

Return the lesser of two numbers

**Synopsis:**

```c
#include <stdlib.h>

#define min(a,b) ...
```

**Arguments:**

`a`, `b`  The numbers that you want to get the lesser of.

**Description:**

The `min()` function returns the lesser of two values.

The `min()` function is for C programs only. For C+ and C++ programs, use the `__max()` or `__min()` macros.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int a;

    a = min( 1, 10 );
    printf( "The value is: %d\n", a );
    return EXIT_SUCCESS;
}
```

**Classification:**

QNX 4

**Safety**

- Cancellation point: No
- Interrupt handler: Yes

continued…

© 2007, QNX Software Systems GmbH & Co. KG.
### min()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

$max()$
Synopsis:
```c
#include <sys/types.h>
#include <sys/stat.h>

int mkdir( const char *path,
           mode_t mode );
```

Arguments:
- `path` The name of the directory that you want to create.
- `mode` The permissions for the directory, modified by the process’s file-creation mask (see `umask()`).

The access permissions for the file or directory are specified as a combination of bits defined in the `<sys/stat.h>` header file. For more information, see “Access permissions” in the documentation for `stat()`.

Library:
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mkdir()` function creates a new subdirectory named `path`. The `path` can be relative to the current working directory or it can be an absolute path name.

The directory’s owner ID is set to the process’s effective user ID. The directory’s group ID is set to the group ID of the parent directory (if the parent set-group ID bit is set) or to the process’s effective group ID.

The newly created directory is empty.

The `mkdir()` function marks the `st_atime`, `st_ctime`, and `st_mtime` fields of the directory for update. Also, the `st_ctime` and `st_mtime` fields of the parent directory are also updated.
mkdir()

Returns:

0, or -1 if an error occurs (errno is set).

Errors:

EACCES Search permission is denied for a component of path, or write permission is denied on the parent directory of path.

EEXIST The directory named by path already exists.

ELOOP Too many levels of symbolic links.

EMLINK The link count of the parent directory would exceed LINK_MAX.

ENAMETOOLONG The length of path exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENOENT A pathname component in the specified path does not exist, or path is an empty string.

ENOSPC The filesystem does not contain enough space to hold the contents of the new directory or to extend the parent directory.

ENOSYS This function is not supported for this path.

ENOTDIR A component of path is not a directory.

EROFS The parent directory resides on a read-only filesystem.

Examples:

To make a new directory called /src in /hd:

#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{ }

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`chdir()`, `chmod()`, `errno`, `getcwd()`, `mknod()`, `rmdir()`, `stat()`, `umask()`
Synopsis:
```c
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo( const char* path,
            mode_t mode );
```

Arguments:
- `path` The pathname that you want to use for the FIFO special file.
- `mode` The file permission bits for the new FIFO. For more information, see “Access permissions” in the documentation for `stat()`.

Library:
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `mkfifo()` function creates a new FIFO special file named by the pathname pointed to by `path`. The file permission bits of the new FIFO are initialized from `mode`, modified by the process’s creation mask (see `umask()`). Bits that are set in `mode` other than the file permission bits are ignored.

The FIFO owner ID is set to the process’s effective user ID and the FIFO’s group ID is set to the process’s effective group ID.

If `mkfifo()` succeeds, the `st_fi`me, `st_c`time, `st_at`ime and `st_m`time fields of the file are marked for update. Also, the `st_c`time and `st_m`time fields of the directory that contains the new entry are marked for update.
Returns:

0    Success.

-1   An error occurred (errno is set).

Errors:

EACCES  A component of the path prefix denies search permission.

EEXIST  The named file already exists.

ENAMETOOLONG  The length of the path string exceeds PATH_MAX, or a
               pathname component is longer than NAME_MAX.

ENOENT   A component of the path prefix doesn’t exist, or the path
         arguments points to an empty string.

ENOSPC   The directory that would contain the new file cannot be extended, or the filesystem is out of file allocation resources (that is, the disk is full).

ENOSYS   This function isn’t supported for this path.

ENOTDIR  A component of the path prefix isn’t a directory.

EROFS    The named file resides on a read-only filesystem.

Examples:

#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    (void)mkfifo( "hd/qnx", S_IRUSR | S_IWUSR );

    return EXIT_SUCCESS;
}
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`chmod()`, `errno`, `mknod()`, `pipe()`, `stat()`, `umask()`
mknod()

© 2007, QNX Software Systems GmbH & Co. KG.

Make a new filesystem entry point

Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>

int mknod( const char * path,  
        mode_t mode,   
        dev_t dev );
```

Arguments:

- **path**: The pathname that you want to use for the file.
- **mode**: A set of bits that define the file type and access permissions that you want to use. The valid file types are:
  - S_IFDIR — create a directory.
  - S_IFIFO — create a FIFO.

For more information, see “Access permissions” in the documentation for `stat()`.

- **dev**: Ignored.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mknod()` makes a file, named `path`, using the file type encoded in the `mode` argument. Supported file types are directories and FIFOs.
This function is included to enhance portability with software written for Unix-compatible operating systems. For POSIX portability, use `mkdir()` or `mkfifo()` instead.

To make a directory with read-write-execute permissions for everyone, you could use the following:

```
  mknod(name, S_IFDIR | 0777, 0);
```

### Returns:

- **0** Success.
- **-1** An error occurred (`errno` is set).

### Errors:

- **EACCES** A component of the path prefix denies search permission, or write permission is denied for the parent directory.
- **EEXIST** The named file already exists.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **EMLINK** The link count of the parent directory would exceed `LINK_MAX`.
- **ENAMETOOLONG** The length of the path string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`.
- **ENOENT** A component of the path prefix doesn’t exist, or the path arguments points to an empty string.
- **ENOSPC** The directory that would contain the new file cannot be extended or the filesystem is out of file allocation resources (that is, the disk is full).
The `mknod()` function isn’t implemented for the filesystem specified in `path`.

A component of the `path` prefix isn’t a directory.

The named file resides on a read-only filesystem.

Examples:

```c
/*
 * Create special files as a directory or FIFO
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main( int argc, char** argv )
{
    int c;
    mode_t mode = 0666;
    int ecode = 0;

    if( argc == 1 ) {
        printf( "Use: %s [-d directory] ... [-f fifo] ... \n",
            argv[0] );
        return( 0 );
    }

    while(( c = getopt( argc, argv, "d:f:" )) != -1 ) {
        switch( c ) {
            case 'd': mode = S_IFDIR | 0666; break;
            case 'f': mode = S_IFIFO | 0666; break;
        }

        if( mknod( optarg, mode, 0 ) != 0 ) {
            perror( optarg );
            ++ecode;
        }
    }

    return( ecode );
}
```
mknod()

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, mkdir(), mkfifo()
**Synopsis:**

```c
#include <stdlib.h>

int mkstemp(char* template);
```

**Arguments:**

`template` A template for the filename that you want to use. This template can be any file name with some number of Xs appended to it, for example `/tmp/temp.XXXX`.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mkstemp()` function takes the given file name template and overwrites a portion of it to create a filename. This file name is unique and suitable for use by the application. The trailing Xs are replaced with the current process number and/or a unique letter combination. The number of unique file names `mkstemp()` can return depends on the number of Xs provided; if you specify six Xs, `mkstemp()` tests roughly $26^6$ combinations.

The `mkstemp()` function (unlike `mktemp()`) creates the template file, mode 0600 (i.e. read-write for the owner), returning a file descriptor opened for reading and writing. This avoids the race between testing for a file’s existence and opening it for use.

**Returns:**

The file descriptor of the temporary file, or -1 if no suitable file could be created; `errno` is set.
Errors:

ENOTDIR The pathname portion of the template isn’t an existing directory.

This function may also set errno to any value specified by open() and stat().

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

It’s possible to run out of letters. The mkstemp() function doesn’t check to determine whether the file name part of template exceeds the maximum allowable filename length.

For portability with X/Open standards prior to XPG4v2, use tmpfile() instead.

See also:

chmod(), getpid(), mktemp(), open() stat(). tmpfile(), tmpnam()
mktemp()

Make a unique temporary filename

Synopsis:

```c
#include <stdlib.h>

char* mktemp( char* template );
```

Arguments:

template A template for the filename that you want to use. This template can be any file name with some number of Xs appended to it, for example /tmp/temp.XXXX.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The mktemp() function takes the given file name template and overwrites a portion of it to create a filename. This file name is unique and suitable for use by the application. The trailing Xs are replaced with the current process number and/or a unique letter combination. The number of unique file names mktemp() can return depends on the number of Xs provided; if you specify six Xs, mktemp() tests roughly $26^6$ combinations.

The mkstemp() function (unlike this function) creates the template file, mode 0600 (i.e. read-write for the owner), returning a file descriptor opened for reading and writing. This avoids the race between testing for a file’s existence and opening it for use.

Returns:

A pointer to the template, or NULL on failure; errno is set.
Errors:

ENOTDIR  The pathname portion of the template isn’t an existing
directory.
This function may also set errno to any value specified by stat().

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

In general, avoid using mktemp(), because a hostile process can
exploit a race condition in the time between the generation of a
temporary filename by mktemp() and the invoker’s use of the
temporary name. Use mkstemp() instead.
This function can create only 26 unique file names per thread for each
unique template.

See also:

chmod(), getpid(), mkstemp(), open() stat(), tmpfile(), tmpnam()
Synopsis:

```c
#include <time.h>

time_t mktime( struct tm* timeptr );
```

Arguments:

- `timeptr` A pointer to a `tm` structure that contains the local time that you want to convert.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mktime()` function converts the local time information in the `struct tm` specified by `timeptr` into a calendar time (Coordinated Universal Time) with the same encoding used by the `time()` function.

The original values of the `tm_sec`, `tm_min`, `tm_hour`, `tm_mday` and `tm_mon` fields aren't restricted to the ranges described for `struct tm`. If these fields aren't in their proper ranges, they're adjusted so that they are. Values for the fields `tm_wday` and `tm_yday` are computed after all the other fields have been adjusted.

The original value of `tm_isdst` is interpreted as follows:

- `< 0` This field is computed as well.
- `0` Daylight savings time isn’t in effect.
- `> 0` Daylight savings time is in effect.

Whenever `mktime()` is called, the `tzset()` function is also called.
Returns:

The converted calendar time, or -1 if mktime() can’t convert it.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

static const char *week_day[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};

int main( void )
{
    struct tm new_year;
    time_t t;

    new_year.tm_year = 2001 - 1900;
    new_year.tm_mon = 0;
    new_year.tm_mday = 1;
    new_year.tm_hour = 0;
    new_year.tm_min = 0;
    new_year.tm_sec = 0;
    new_year.tm_isdst = 0;

    t = mktime( &new_year );
    if ( t == (time_t)-1)
        printf("No conversion possible.\n");
    else
        printf("The 21st century begins on a %s.\n",
               week_day[ new_year.tm_wday ] );

    return EXIT_SUCCESS;
}
```

produces the output:

The 21st century begins on a Monday.

Classification:

ANSI, POSIX 1003.1
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), clock(), ctime(), ctime_r(), difftime(),
gmtime(), gmtime_r(), localtime(), localtime_r(), strftime(), time(),
tm, tzset()
mlock()
Lock a range of process address space in physical memory

Synopsis:

```c
#include <sys/mman.h>

int mlock(const void * addr,
           size_t len);
```

Arguments:

- `addr` The starting address for the range of process address space.
- `len` Length of the memory buffer in bytes.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mlock()` function locks a range of process address space starting at address `addr` and continuing for length `len`. The `addr` must be a multiple of `PAGESIZE`, which depends on the target platform.

The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.

The successful call to `mlock()` function ensures memory-resident pages.

The calling process needs superuser capabilities to call the `mlock()` function.

You always map the memory (using `mmap()`) region in three locking states:
mlock() © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>State</th>
<th>Result of memory mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>May be paged in/out</td>
</tr>
<tr>
<td>Locked</td>
<td>May not be paged in/out. May still fault on access/reference to maintain usage/modification stats</td>
</tr>
<tr>
<td>Superlocked</td>
<td>Happens when I/O privileges are granted to a thread. No faulting allowed at all and covers the whole address space</td>
</tr>
</tbody>
</table>

For MAP_LAZY mappings, memory is not allocated/mapped until the memory is first referenced for any of the above types. Once it’s been referenced, it obeys the above rules — it’s a programmer error to touch a MAP_LAZY area in a critical region (interrupts disabled or an ISR) that hasn’t already been referenced.

The default locking state of memory is determined by the -m[~]L and -m[~]L options of the procnto manager.

Memory-resident is a term used to indicate that the addresses always reside in physical memory.

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

ENOMEM • Some or all of the address range specified by the addr and len arguments doesn’t correspond to valid mapped pages in the address space of the process.

• Locking the pages mapped by the specified range would exceed an implementation-defined limit on the amount of memory that the process may lock.
This implementation-defined limit is set by
RLIMIT_MEMLOCK rlimit.

EAGAIN Some or all of the memory identified by the operation
couldn’t be locked when the call was made.

EPERM The calling process doesn’t have the superuser
capabilities.

Classification:

POSIX 1003.1 MLR

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mlockall(), mmap(), munlock()
**mlockall()**

Lock all of a process’s address space

**Synopsis:**

```c
#include <sys/mman.h>

int mlockall(int flags);
```

**Arguments:**

- `flags` Flags that indicate which pages to lock; one or more of the following bits:
  
  - **MCL_CURRENT** Lock the pages currently mapped into the address space of the process.
  - **MCL_FUTURE** Lock the pages that become mapped into the address space of the process in the future, when the mappings are established.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mlockall()` function causes all of the pages mapped by the address space of a process to be locked and made memory-resident until unlocked or the process exits or executes another process and is determined by the `flags` argument.

---

The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.
Memory-resident is a term used to indicate that the addresses always reside in physical memory.

You always map the memory (using `mmap()`) region in three locking states:

<table>
<thead>
<tr>
<th>State</th>
<th>Result of memory mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocked</td>
<td>May be paged in/out</td>
</tr>
<tr>
<td>Locked</td>
<td>May not be paged in/out. May still fault on access/reference to maintain usage/modification stats</td>
</tr>
<tr>
<td>Superlocked</td>
<td>Happens when I/O privileges are granted to a thread. No faulting allowed at all and covers the whole address space</td>
</tr>
</tbody>
</table>

For MAP_LAZY mappings, memory is not allocated/mapped until the memory is first referenced for any of the above types. Once it’s been referenced, it obeys the above rules — it’s a programmer error to touch a MAP_LAZY area in a critical region (interrupts disabled or an ISR) that hasn’t already been referenced.

The default locking state of memory is determined by the `-m[~]l` and `-m[~]L` options of the `procnto` manager.

You have to be a superuser to lock pages. Follow either of the following approaches when attempting to lock pages:

Tightly controlled approach

- Lock the pages one by one, by calling `mmap()` and then `mlock()`.

Global approach

- Lock all pages at the same time, by doing one of the following:
mlockall() © 2007, QNX Software Systems GmbH & Co. KG.

- Call mlockall(MCL_FUTURE), followed by one or more calls to mmap()
- Optionally call mmap(), and then call mlockall(MCL_CURRENT)

Returns:

  0 Success.
  -1 An error occurred (errno is set).

Errors:

EAGAIN Some or all of the memory identified by the operation couldn’t be locked when the call was made.

EINVAL The flags argument is zero.

ENOMEM Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock. This implementation-defined limit is set by RLIMIT_MEMLOCK rlimit.

EPERM The calling process doesn’t have the appropriate privilege to perform the requested operation.

Classification:

POSIX 1003.1 ML

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
mlockall()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mlock(), mmap(), munlock()
**mmap(), mmap64()**

Map a memory region into a process’s address space

**Synopsis:**

```c
#include <sys/mman.h>

void * mmap( void * addr,
             size_t len,
             int prot,
             int flags,
             int fildes,
             off_t off );

void * mmap64( void * addr,
               size_t len,
               int prot,
               int flags,
               int fildes,
               off64_t off );
```

**Arguments:**

- **addr** NULL, or a pointer to where you want the object to be mapped in the calling process’s address space.
- **len** The number of bytes to map into the caller’s address space. It can’t be 0.
- **prot** The access capabilities that you want to use for the memory region being mapped. You can combine at least the following protection bits, as defined in `<sys/mman.h>`:
  - PROT_EXEC — the region can be executed.
  - PROT_NOCACHE — disable caching of the region (e.g. so it can be used to access dual-ported memory).
  - PROT_NONE — the region can’t be accessed.
  - PROT_READ — the region can be read.
  - PROT_WRITE — the region can be written.
- **flags** Flags that specify further information about handling the mapped region; see below.
mmap(), mmap64()

fildes  The file descriptor for a shared memory object, or NOFD if you’re mapping physical memory.

off    The offset into shared memory of the location that you want to start mapping.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The mmap() function maps a region within the object beginning at off and continuing for len into the caller’s address space and returns the location.

Typically, you don’t need to use addr; you can just pass NULL instead. If you set addr to a non-NULL value, whether the object is mapped depends on whether or not you set MAP_FIXED in flags:

MAP_FIXED is set
    The object is mapped to the address in addr, or the function fails.

MAP_FIXED isn’t set
    The value of addr is taken as a hint as to where to map the object in the calling process’s address space. The mapped area won’t overlay any current mapped areas.

There are two parts to the flags parameter. The first part is a type (masked by the MAP_TYPE bits), which you must specify as one of the following:

MAP_PRIVATE
    The mapping is private to the calling process. It allocates system RAM and copies the current object.
MAP_SHARED
The mapping may be shared by many processes.

You can OR the following flags into the above type to further specify the mapping:

MAP_ANON This is most commonly used with MAP_PRIVATE. The fildes parameter must be NOFD. The allocated memory is zero-filled. This is equivalent to opening /dev/zero.

MAP_BELOW16M Used with MAP_PHYS | MAP_ANON. The allocated memory area resides in physical memory below 16M. This is important for using DMA with ISA bus devices.

MAP_FIXED Map the object to the address specified by addr. If this area is already mapped, the call changes the existing mapping of the area.

Use MAP_FIXED with caution. Not all memory models support it. In general, you should assume that you can MAP_FIXED only at an address (and size) that a call to mmap() without MAP_FIXED returned.

A memory area being mapped with MAP_FIXED is first unmapped by the system using the same memory area. See munmap() for details.

MAP_LAZY Delay acquiring system memory, and copying or zero-filling the MAP_PRIVATE or MAP_ANON pages, until an access to the area has occurred. If you set this flag, and there’s no system memory at the time of the access, the thread gets a SIGBUS with a code of BUS_ADRERR. This flag is a hint to the memory manager.
mmap(), mmap64()

MAP_PHYS  Physical memory is required. The fildes parameter must be NOFD. When used with MAP_PRIVATE or MAP_SHARED, the offset specifies the exact physical address to map (e.g. for video frame buffers), and is equivalent to opening /dev/mem. If used with MAP_ANON, then physically contiguous memory is allocated.

MAP_NOX64K and MAP_BELOW16M are used to further define the MAP_ANON allocated memory (useful on x86 only).

You should use mmap_device_memory() instead of MAP_PHYS.

MAP_NOINIT  When specified, the POSIX requirement that the memory be zeroed is relaxed. The physical memory being used for this allocation must have been previously freed with UNMAP_INIT_OPTIONAL for this flag to have any effect.

This flag was added in the QNX Neutrino Core OS 6.3.2.

MAP_NOX64K  (Useful on x86 only). Used with MAP_PHYS | MAP_ANON. Prevent the allocated memory area from crossing a 64K boundary. This may be important to some DMA devices. If more than 64K is requested, the area begins on a 64K boundary.

MAP_STACK  This flag tells the memory allocator what the MAP_ANON memory will be used for. It’s only a hint.

Using the mapping flags described above, a process can easily share memory between processes:
/* Map in a shared memory region */
fd = shm_open( "/datapoints", O_RDWR, 0777 );
addr = mmap( 0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0 );

To share memory with hardware such as video memory on an x86 platform:

/* Map in VGA display memory */
addr = mmap( 0,
65536,
PROT_READ|PROT_WRITE,
MAP_PHYS|MAP_SHARED,
NOFD,
0xa0000 );

To allocate a DMA buffer for a bus-mastering PCI network card:

/* Allocate a physically contiguous buffer */
addr = mmap( 0,
262144,
PROT_READ|PROT_WRITE|PROT_NOCACHE,
MAP_PHYS|MAP_ANON,
NOFD,
0 );

Returns:
The address of the mapped-in object, or MAP_FAILED if an error occurred (errno is set).

Errors:

EACCES The file descriptor in fildes isn’t open for reading, or you specified PROT_WRITE and MAP_SHARED, and fildes isn’t open for writing.

EBADF Invalid file descriptor, fildes.

EINVAL Invalid flags type, or len is 0.

ENODEV The fildes argument refers to an object for which mmap() is meaningless (e.g. a terminal).
mmap(), mmap64()

ENOMEM You specified MAP_FIXED, and the address range requested is outside of the allowed process address range, or there wasn’t enough memory to satisfy the request.

ENXIO The address from off for len bytes is invalid for the requested object, or you specified MAP_FIXED, and addr, len, and off are invalid for the requested object.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(int argc, char *argv[]) {
    int i;
    unsigned char *addr, c;
    
    /* Map BIOS ROM */
    addr = mmap(0, 0x10000, PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_PHYS, NOFD, 0xf0000);
    if (addr == MAP_FAILED) {
        fprintf(stderr, "mmap failed : %s\n",
                strerror(errno));
        return EXIT_FAILURE;
    }
    printf("Map addr is %p\n",(void*) addr);
    
    for (i = 0; i < 3 * 80; ++i) {
        c = *addr++;
        if (c >= ' ' && c <= 0x7f)
            putchar(c);
        else
            putchar('.');
    }
    
    return EXIT_SUCCESS;
}
```
**mmap()**, **mmap64()**

### Classification:

*mmap()* is POSIX 1003.1 MF | SHM | TYM; *mmap64()* is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

*mmap_device_io()*, *mmap_device_memory()*, *munmap()*
Synopsis:

```c
#include <stdint.h>
#include <sys/mman.h>

uintptr_t mmap_device_io( size_t len,
                            uint64_t io );
```

Arguments:

- `len` The number of bytes of device I/O memory that you want to access. It can’t be 0.
- `io` The address of the area that you want to access.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mmap_device_io()` function maps `len` bytes of device I/O memory at `io` and makes it accessible via the `in*()` and `out*()` functions in `<hw/inout.h>`.

Returns:

A handle to the device’s I/O memory, or MAP_DEVICE_FAILED if an error occurs (errno is set).

Errors:

- **EINVAL** Invalid flags type, or `len` is 0.
- **ENOMEM** The address range requested is outside of the allowed process address range, or there wasn’t enough memory to satisfy the request.
- **ENXIO** The address from `io` for `len` bytes is invalid.
**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

You need I/O privileges to use the result of the `mmap_device_io()` function. The calling thread may call `ThreadCtl()` with the _NTO_TCTL_IO command to establish these privileges.

**See also:**

`mmap()`, `mmap_device_memory()`, `munmap_device_io()`
**mmap_device_memory()**

Map a device's physical memory into a process's address space

**Synopsis:**
```
#include <sys/mman.h>

void * mmap_device_memory( void * addr,
                          size_t len,
                          int prot,
                          int flags,
                          uint64_t physical );
```

**Arguments:**
- **addr**
  - NULL, or a pointer to where you want to map the object in the calling process's address space.
- **len**
  - The number of bytes you want to map into the caller's address space. It can't be 0.
- **prot**
  - The access capabilities that you want to use for the memory region being mapped. You can use a combination of at least the following protection bits, as defined in `<sys/mman.h>`:
    - PROT_EXEC — the region can be executed.
    - PROT_NOCACHE — disable the caching of the region (e.g. to access dual-ported memory).
- **flags**
  - Specifies further information about handling the mapped region. You can use the following flag:
    - PROT_NONE — the region can't be accessed.
    - PROT_READ — the region can be read.
    - PROT_WRITE — the region can be written.

Read the architecture guide for your processor; you may need to add special instructions. For example, if you specify PROT_NOCACHE on a PPC device, you may need to issue special `eieio` (Enforce In-Order Execution of I/O) instructions to ensure that writes occur in a desired order. You'll find an `eieio()` macro in `<ppc/inout.h>`.

September 10, 2007
**mmap_device_memory()**

- MAP_FIXED — map the object to the address specified by `addr`. If this area is already mapped, the call changes the existing mapping of the area.

Use MAP_FIXED with caution. Not all memory models support it. In general, you should assume that you can MAP_FIXED only at an address (and size) that a call to `mmap()` without MAP_FIXED returned.

A memory area being mapped with MAP_FIXED is first unmapped by the system using the same memory area. See `munmap()` for details.

This function already uses MAP_SHARED ORed with MAP_PHYS (see `mmap()` for a description of these flags).

`physical` The physical address of the memory to map into the caller’s address space.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mmap_device_memory()` function maps `len` bytes of a device’s physical memory address into the caller’s address space at the location returned by `mmap_device_memory()`.

You should use this function instead of using `mmap()` with the MAP_PHYS flag.

Typically, you don’t need to use `addr`; you can just pass NULL instead. If you set `addr` to a non-NULL value, whether the object is mapped depends on whether or not you set MAP_FIXED in `flags`:
mmap_device_memory()

MAP_FIXED is set
The object is mapped to the address in addr, or the function fails.

MAP_FIXED isn’t set
The value of addr is taken as a hint as to where to map the object in the calling process’s address space. The mapped area won’t overlay any current mapped areas.

Returns:
The address of the mapped-in object, or MAP_FAILED if an error occurs (errno is set).

Errors:
EINVAL Invalid flags type, or len is 0.
ENOMEM The address range requested is outside of the allowed process address range, or there wasn’t enough memory to satisfy the request.
ENXIO The address from physical for len bytes is invalid for the requested object, or MAP_FIXED was specified and addr, len, and physical are invalid for the requested object.

Examples:
/* map in the physical memory, 0xb8000 is text mode VGA video memory */
ptr = mmap_device_memory( 0, len, PROT_READ|PROT_WRITE|PROT_MOCACHE, 0, 0xb8000 );
if ( ptr == MAP_FAILED ) {
    perror( "mmap_device_memory for physical address 0xb8000 failed" );
    exit( EXIT_FAILURE );
}

Classification:
QNX Neutrino
mmap_device_memory()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mmap(), mmap_device_io(), munmap_device_memory()
Synopsis:

```c
#include <sys/modem.h>

int modem_open( char* device,
                speed_t baud );
```

Arguments:

- `device` - The path name of the serial port that you want to open.
- `baud` - Zero, or the baud rate that you want to use.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `modem_open()` function opens a serial port identified by `device`. The device is set to raw mode by changing the control flags using `tcgetattr()` and `tcsetattr()` as follows:

```c
termio.c_cflag = CS8|INFLOW|OHFLOW|CREAD|HUPCL;
termio.c_iflag = BRKINT;
termio.c_lflag = IEXTEN;
termio.c_oflag = 0;
```

Any pending input or output characters are discarded.

If `baud` is nonzero, then the baud rate is changed to that value.
modem_open()

Returns:

An open file descriptor, or -1 on failure (errno is set).

Errors:

EACCES  Search permission is denied on a component of the path prefix, or the file doesn’t exist.

EBADFSYS  While attempting to open the named file, either the file itself or a component of the path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You’ll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY  The file named by device is a block special device that’s already open for writing, or device names a file that’s on a filesystem mounted on a block special device that’s already open for writing, or device is in use.

EINTR  The open operation was interrupted by a signal.

EISDIR  The named device is a directory.

ELOOP  Too many levels of symbolic links or prefixes.

EMFILE  Too many file descriptors are currently in use by this process.

ENAMETOOLONG  The length of the device string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENFILE  Too many files are currently open in the system.

ENOENT  The named device doesn’t exist, or the path argument points to an empty string.
The \texttt{modem\_open()} function isn't implemented for the filesystem specified in \texttt{device}.

A component of the path prefix isn't a directory.

No process has the file open for reading.

\textbf{Examples:}

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/modem.h>
#include <stdio.h>
#include <errno.h>

/*
curstate curflags newstate newflags newtimeout
newquiet retvalue pattern response
*/

struct modem_script table[] = {
    {1, 0, 1, 0, 2, 5, 0, NULL, "ATZ\r\P0a"},
    {1, 0, 2, 0, 30, 5, 0, "*ok*", "ATDTE5910934"},
    {2, MODEM_BAUD, 3, MODEM_LASTLINE, 10, 5, 0, "*connect*", NULL},
    {3, 0, 4, 0, 8, 5, 0, "*login:*", "guest"},
    {4, MODEM_NOECHO, 5, 0, 15, 5, 0, "*password:*", "xxxx"},
    {5, 0, 0, 0, 0, 0, 0, NULL},
    {0, 0, 0, 0, 0, 0, 0, NULL},
    {0, 0, 0, 0, 0, 0, 1, NULL},
    {0, 0, 0, 0, 0, 0, 2, NULL},
    {0, 0, 0, 0, 0, 0, 3, NULL},
    {0, 0, 0, 0, 0, 0, 4, NULL},
    { NULL }
};

void io(char* progress, char* in, char* out) {
    if(progress)
```

September 10, 2007

C Library — M to O

1693
# modem_open()

```c
printf("progress: %s\n", progress);

if(in)
    printf("input: %s\n", in);

if(out)
    printf("output: %s\n", out);
}

int main(int argc, char* argv[]) {
    int fd, status;
    speed_t baud = -1;

    if((fd = modem_open(argv[1], 0)) == -1) {
        fprintf(stderr, "Unable to open %s: %s\n",
            argv[1], strerror(errno));
        exit(1);
    }

    status = modem_script( fd, table, &baud,
        &io, NULL );
    printf("status=%d baud=%d\n", status, baud);
    exit(status);
}
```

### Classification:

**QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

modem_read(), modem_script(), modem_write()
modem_read()

Read bytes from a file descriptor

Synopsis:

```c
#include <sys/modem.h>

int modem_read( int fd,
    char* buf,
    int bufsize,
    int quiet,
    int timeout,
    int flags,
    int (*cancel)(void) );
```

Arguments:

- `fd` The file descriptor for the device that you want to read from; see `modem_open()`.
- `buf` A pointer to a buffer where the function can store the data.
- `bufsize` The size of the buffer, in bytes.
- `quiet` The maximum time to wait for more input after receiving at least one characters, in tenths of a second.
- `timeout` The maximum time to wait for any input, in tenths of a second.
- `flags` Flags that you can use to filter and map received characters; any combination of:
  - `MODEM_ALLOWCASE` — preserve the case of incoming characters. Without this flag, all letters are mapped to lower case.
  - `MODEM_ALLOWCTRL` — allow control characters. Without this flag, control characters are discarded.
  - `MODEM_ALLOW8BIT` — preserve the top bit of incoming characters. Without this flag, the top bit is set to zero for all characters.
  - `MODEM_LASTLINE` — discard all previously received characters when a newline is received.
modem_read()

followed by more characters. Without this flag, \texttt{buf}
may contain multiple lines. If an automatic login
script may be presented with an arbitrary text screen
before the login prompt, you can use this flag to
discard all but the login line, reducing the possibility
of false matches.

\texttt{cancel} \ NULL, or a callback that’s called whenever the \texttt{quiet} time
period expires while waiting for more input.

Library:

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is
usually included automatically.

This function is in \texttt{libc.a}, but not in \texttt{libc.so} (in order to save
space).

Description:

The \texttt{modem_read()} function reads up to \texttt{bufsize} bytes from the device
specified by the file descriptor, \texttt{fd}, and places them into the buffer
pointed to by \texttt{buf}.

If no characters are received within the given \texttt{timeout}, \texttt{modem_read()}
returns with \texttt{-1}.

When at least one character has been received, \texttt{modem_read()} returns
if the flow of incoming characters stops for at least the \texttt{quiet} time
period. The number of characters saved in \texttt{buf} is returned.

If you provide a \texttt{cancel} function, it’s called once each \texttt{quiet} time
period while waiting for input. If this function returns a nonzero
value, \texttt{modem_read()} returns \texttt{-1} immediately and sets \texttt{errno} to
\texttt{ETIMEDOUT}. You can use the \texttt{cancel} function as a callback in a
graphical dialer that needs to support a cancel button to stop a script
(see \texttt{modem_script()}).
**modem_read()**

Returns:

Zero for success, or -1 on failure (`errno` is set).

Errors:

- **EAGAIN** The `O_NONBLOCK` flag is set on this `fd`, and the process would have been blocked in trying to perform this operation.
- **EBADF** The argument `fd` is invalid, or the file isn’t opened for reading.
- **EINVAL** The `readcond()` call was interrupted by the process being signalled.
- **EIO** This process isn’t currently able to read data from this `fd`.
- **ENOSYS** This function isn’t supported for this `fd`.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancelation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Read the Caveats</td>
</tr>
</tbody>
</table>

**Caveats:**

Depending on what you do in your `cancel` function, `modem_read()` may or not be signal handler or thread-safe.
See also:

`modem_open()`, `modem_script()`, `modem_write()`
modem_script()
© 2007, QNX Software Systems GmbH & Co. KG.
Run a script on a device

Synopsis:
#include <sys/modem.h>

int modem_script( int fd,  
    struct modem_script* table,  
    speed_t* baud,  
    void (*io)(  
        char* progress,  
        char* in,  
        char* out),  
    int (*cancel)(void) );

Arguments:

    fd       The file descriptor for the device that you want to read  
            from; see modem_open().
    table    An array of modem_script structures that comprise a  
            script of commands that you want to run on the device; see  
            below.
    baud     A pointer to a speed_t where the function can store the  
            baud rate (if you script says to do so).
    io       A function that’s called to process each string that’s  
            emitted or received.
    cancel   NULL, or a callback function that’s called whenever the  
            newquiet time period (specified in the script) expires while  
            waiting for input.

Library:

    libc
    Use the -l c option to qcc to link against this library. This library is  
    usually included automatically.
This function is in \texttt{libc.a}, but not in \texttt{libc.so} (in order to save space).

Description:

The \texttt{modem_script()} function runs the script \texttt{table} on the device associated with the file descriptor \texttt{fd}. The script implements a simple state machine that emits strings and waits for responses.

Each string that’s emitted or received is passed to the function \texttt{io()} as follows:

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{(\ast io)(str, 0, 0)}</td>
<td>Emitted \textit{progress} string</td>
</tr>
<tr>
<td>\texttt{(\ast io)(0, str, 0)}</td>
<td>Received string</td>
</tr>
<tr>
<td>\texttt{(\ast io)(0, 0, str)}</td>
<td>Emitted \textit{response} string</td>
</tr>
</tbody>
</table>

This lets an application set up a callback that can display the script’s interaction in a status window.

If you provide a \texttt{cancel} function, it’s called once each \texttt{newquiet} 1/10 of a second while waiting for input. If this function returns a nonzero value, the read returns immediately with -1 and \texttt{errno} is set to \texttt{ETIMEDOUT}. You can use the \texttt{cancel} function as a callback in a graphical dialer that needs to support a cancel button to stop a script.

The \texttt{table} is an array of \texttt{modem_script} structures that contain the following members:

- \texttt{char curstate} The current state. Execution always begins at state 1, which must be the first array element of \texttt{table}. Multiple elements may have the same current state.
state, in which case any received input is matched against each pattern member for that state.

**int curflags**  
The flags to use on a pattern match of a response:

- MODEM_NOECHO — don’t echo the response through the io() callback.
- MODEM_BAUD — extract any number in the response and assign it to baud.

**char newstate**  
When a pattern match occurs with pattern, this is the next state. A state transition causes response to be output and newflags, newtimeout, and newquiet to be saved and associated with the new state. Changing to a new state of 0 causes modem_script() to return with the value in retvalue.

**int newflags**  
Saved on a state transition and passed to modem_read() when waiting for a response in the new state. For information about these flags, see modem_read().

**int newtimeout**  
Saved on a state transition and passed to modem_read() when waiting for a response in the new state. This timeout is described in modem_read().

**int newquiet**  
Saved on a state transition and passed to modem_read() when waiting for a response in the new state. This quiet timeout is described in modem_read().

**short retvalue**  
The return value when the script terminates with a pattern match, and the new state is 0.

**char* pattern**  
A pattern to match against received characters. The pattern is matched using fnmatch(). Only patterns in the current state or the wildcard state of 0 are matched. On a match, the current state changes to newstate.
char* response  On a pattern match, this response is output to the device. If the curflags don’t have MODEM_NOECHO set, the response is given to the callback function passed as the io parameter.

char* progress  On a pattern match, this progress string is passed to the callback function passed as the io parameter.

Here’s an example that demonstrates the operation of the script:

```c
/*
curstate curflags newstate newflags newtimeout
newquiet retvalue pattern response
*/

struct modem_script table[] ={  
{1, 0, 1, 0, 2, 5, 0,  
 NULL, "ATZ\r\P0a"},
{1, 0, 2, 0, 30, 5, 0,  
 "*ok*", "ATDT5910934"},
{2, MODEM_BAUD, 3, MODEM_LASTLINE, 10, 5, 0,  
 "*connect*", NULL},
{3, 0, 4, 0, 8, 5, 0,  
 "*login*", "guest"},
{4, MODEM_NOECHO, 5, 0, 15, 5, 0,  
 "*password*", "xxxx"},
{5, 0, 0, 0, 0, 0, 0, 0,  
 "*$ *", NULL},
{0, 0, 0, 0, 0, 1,  
 "*no carrier*", NULL},
{0, 0, 0, 0, 2,  
 "*no answer*", NULL},
{0, 0, 0, 0, 3,  
 "*no dialtone*", NULL},
{0, 0, 0, 0, 4,  
 "*busy*", NULL},
{ NULL }
};
```

When this script is passed to `modem_script()`, the current state is set to 1, and the output is `ATZ` (the response in the first array element).

While in any state, `modem_script()` waits for input, matching it against the current state or the wildcard state of 0.
### State 1

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ok</em></td>
<td>Go to state 2 and emit <code>ATDT1-591-0934</code>. The flags to be used in the new state are set to 0, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 30 seconds.</td>
</tr>
<tr>
<td><em>no carrier</em></td>
<td>Go to state 0 (the termination newstate), return with the contents of <code>retval</code> (1).</td>
</tr>
<tr>
<td><em>no answer</em></td>
<td>Go to state 0 (the termination newstate), return with the contents of <code>retval</code> (2).</td>
</tr>
<tr>
<td><em>no dialtone</em></td>
<td>Go to state 0 (the termination newstate), return with the contents of <code>retval</code> (3).</td>
</tr>
<tr>
<td><em>busy</em></td>
<td>Go to state 0 (the termination newstate), return with the contents of <code>retval</code> (4).</td>
</tr>
</tbody>
</table>

### State 2

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>connect</em></td>
<td>Go to state 3 and don’t emit anything to the device. The flags to be used in the new state are set to MODEM_LASTLINE, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 10 seconds. Since the current flags are MODEM_BAUD, the baud rate is extracted from the connect message.</td>
</tr>
<tr>
<td><em>no carrier</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>no answer</em></td>
<td>Same as previous table</td>
</tr>
</tbody>
</table>

*continued...*
### modem_script()

**State 3**

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>login</em></td>
<td>Go to state 4 and emit guest. The flags to be used in the new state are set to 0, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 8 seconds.</td>
</tr>
<tr>
<td><em>no carrier</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>no answer</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>no dialtone</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>busy</em></td>
<td>Same as previous table</td>
</tr>
</tbody>
</table>

**State 4**

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>password</em></td>
<td>Go to state 5 and emit xxxx. The flags to be used in the new state are set to 0, the quiet time in the new state is set to 5/10 of a second, and the timeout time in the new state is set to 15 seconds. Since the current flags are MODEM_NOECHO, the password response xxxx isn’t sent to the io() callback.</td>
</tr>
<tr>
<td><em>no carrier</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>no answer</em></td>
<td>Same as previous table</td>
</tr>
</tbody>
</table>

continued...
**modem_script()**

© 2007, QNX Software Systems GmbH & Co. KG.

### Input | Action
---|---
*no dialtone* | Same as previous table
*busy* | Same as previous table

#### State 5

<table>
<thead>
<tr>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>$</em></td>
<td>Go to state 0 (the termination newstate), return with the contents of <em>retval</em> (0).</td>
</tr>
<tr>
<td><em>no carrier</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>no answer</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>no dialtone</em></td>
<td>Same as previous table</td>
</tr>
<tr>
<td><em>busy</em></td>
<td>Same as previous table</td>
</tr>
</tbody>
</table>

If you set the flag MODEM_BAUD for a state, then any number embedded in a matching response is extracted and assigned as a number to the **baud** parameter.

If you don’t set the flag MODEM_NOECHO for a state, then all emitted strings are also given to the passed **io** function as (*io*)(0, 0, *response*).

**Returns:**

The *retval* member of a script entry that terminates the script. This will always be a positive number. If *modem_script* fails, it returns -1 and sets *errno*.

**Errors:**

- **EAGAIN** The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the write operation.
- **EBADF** The file descriptor, *fildes*, isn’t a valid file descriptor open for writing.
modem_script()

EINTR  The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.

EIO    A physical I/O error occurred. The precise meaning depends on the device.

EPIPE  An attempt was made to write to a pipe (or FIFO) that isn’t open for reading by any process. A SIGPIPE signal is also sent to the process.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Read the Caveats</td>
</tr>
</tbody>
</table>

Caveats:

Depending on what you do in your cancel function, it might or might not be safe to call modem_script() from a signal handler or a multithreaded program.

See also:

modem_open(), modem_read(), modem_write()
modem_write()

Write a string to a device

Synopsis:

```c
#include <sys/modem.h>

int modem_write( int fd,
    char* str );
```

Arguments:

- `fd` The file descriptor for the device that you want to write to; see `modem_open()`.
- `str` The string that you want to write.

Library:

- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `modem_write()` function writes the string `str` to the device specified by the file descriptor `fd`. Just before writing each character, all buffered input from the same device is flushed. After writing each character, an attempt to read an echo is made. The intent is to write a string without its appearing back in the input stream even if the device is echoing each character written.

If the `\` character appears in `str`, then the character following it is interpreted by `modem_write()`, and instead of both being written, they’re treated as a special escape sequence that causes the following actions to be taken:
# modem_write()

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r</td>
<td>Output a carriage return.</td>
</tr>
<tr>
<td>\n</td>
<td>Output a newline.</td>
</tr>
<tr>
<td>\xhh</td>
<td>Output the single character whose hex representation follows as hh.</td>
</tr>
<tr>
<td>\B</td>
<td>Send a 500 msec break on the line using tcsendbreak().</td>
</tr>
<tr>
<td>\D</td>
<td>Drop the line for 1 second using tcdropline().</td>
</tr>
<tr>
<td>\Phh</td>
<td>Pause for hh 1/10 of a second where hh is two hex characters.</td>
</tr>
</tbody>
</table>

## Returns:

Zero on success, -1 on failure (errno is set).

## Errors:

- **EAGAIN**: The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the write operation.
- **EBADF**: The file descriptor, fildes, isn’t a valid file descriptor open for writing.
- **EINTR**: The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.
- **EIO**: A physical I/O error occurred. The precise meaning depends on the device.
- **EPIPE**: An attempt was made to write to a pipe (or FIFO) that isn’t open for reading by any process. A SIGPIPE signal is also sent to the process.
**modem_write()**

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*modem_open(), modem_read(), modem_script()*
Synopsis:

```c
#include <math.h>

double modf( double value,
             double* iptr);

float modff( float value,
             float* iptr);
```

Arguments:

- `value`: The value that you want to break into parts.
- `iptr`: A pointer to a location where the function can store the integral part of the number.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `modf()` and `modff()` functions break the given `value` into integral and fractional parts, each of which has the same sign as the argument. They store the integral part as a `double` in the object pointed to by `iptr`.

Returns:

The signed fractional part of `value`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    double integral_value, fractional_part;

    fractional_part = modf( 4.5, &integral_value );
    printf( "%f %f\n", fractional_part, integral_value );

    fractional_part = modf( -4.5, &integral_value );
    printf( "%f %f\n", fractional_part, integral_value );

    return EXIT_SUCCESS;
}
```
produces the output:
```
0.500000 4.000000
-0.500000 -4.000000
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`frexp()`, `ldexp()`
mount()
Mount a filesystem

Synopsis:
#include <sys/mount.h>

int mount( const char* spec,
            const char* dir,
            int flags,
            const char* type,
            const void* data,
            int datalen );

Arguments:

spec A null-terminated string describing a special device (e.g. /dev/hd0t77), or NULL if there’s no special device.
dir A null-terminated string that names the directory that you want to mount (e.g. /mnt/home).
flags Flags that are passed to the driver:
  • _MFLAG_OCB — ignore the special device string, and contact all servers.
  • _MOUNT_READONLY — mark the filesystem mountpoint as read-only.
  • _MOUNT_NOEXEC — don’t allow executables to load.
  • _MOUNT_NOSUID — don’t honor setuid bits on the filesystem.
  • _MOUNT_NOCREAT — don’t allow file creation on the filesystem.
  • _MOUNT_OFF32 — limit off_t to 32 bits.
  • _MOUNT_NOATIME — disable logging of file access times.
  • _MOUNT_BEFORE — call resmgr_attach() with RESMGR_FLAG_BEFORE.
  • _MOUNT_AFTER — call resmgr_attach() with RESMGR_FLAG_AFTER.
mount() — call resmgr_attach() with RESMGR_FLAG_OPAQUE.

- _MOUNT_UNMOUNT — unmount this path.
- _MOUNT_REMOUNT — this path is already mounted; perform an update.
- _MOUNT_FORCE — force an unmount or a remount change.
- _MOUNT_ENUMERATE — autodetect on this device.

**type**
A null-terminated string with the filesystem type (e.g. nfs, cifs, qnx4, ext2, network).

**data**
A pointer to additional data to be sent to the manager. If datalen is <0, the data points to a null-terminated string.

**datalen**
The length of the data, in bytes, that’s being sent to the server, or <0 if the data is a null-terminated string.

**Library:**

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `mount()` function sends a request to servers to mount the services provided by `spec` and `type` at `dir`.

If you set _MFLAG_OCB in the flags, then the special device string is ignored, and all servers are contacted. If you don’t set this bit, and the special device `spec` exists, then only the server that created that device is contacted, and the full path to `spec` is provided.

If `datalen` is any value <0, and there’s a data pointer, the function assumes that the data pointer is a pointer to a string.
mount()

Returns:
-1 on failure; no server supports the request (errno is set).

Classification:
QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
resmgr_attach(), umount()

Writing a Resource Manager in Programmer’s Guide
### mount_parse_generic_args() © 2007, QNX Software Systems GmbH & Co. KG.

**Synopsis:**

```c
#include <sys/mount.h>

char * mount_parse_generic_args( char * options,
                                 int * flags );
```

**Arguments:**

- **options**: The string of options that you want to parse; see below.
- **flags**: A pointer to a location where the function can store a set of bits corresponding to the options that it finds; see below.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `mount_parse_generic_args()` function allows you to strip out common flags to help you parse mount arguments. This is useful when you want to create a custom mount utility.

Here's a list of the supported options that may be stripped:

<table>
<thead>
<tr>
<th>Option</th>
<th>Set/Clear this bit:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>after</td>
<td>Set _MOUNT_AFTER</td>
<td>Call <code>resmgr_attach()</code> with RESMGR_FLAG_AFTER.</td>
</tr>
<tr>
<td>atime</td>
<td>Clear _MOUNT_ATIME</td>
<td>Log file access times (default).</td>
</tr>
<tr>
<td>before</td>
<td>Set _MOUNT_BEFORE</td>
<td>Call <code>resmgr_attach()</code> with RESMGR_FLAG_BEFORE.</td>
</tr>
</tbody>
</table>

(...continued...)
<table>
<thead>
<tr>
<th>Option</th>
<th>Set/Clr this bit:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creat</td>
<td>Clear _MOUNT_CREAT</td>
<td>Allow file creation on the filesystem (default).</td>
</tr>
<tr>
<td>enumerate</td>
<td>Set _MOUNT_ENUMERATE</td>
<td>Auto-detect on this device.</td>
</tr>
<tr>
<td>exec</td>
<td>Clear _MOUNT_NOEXEC</td>
<td>Load executables (default).</td>
</tr>
<tr>
<td>force</td>
<td>Set _MOUNT_FORCE</td>
<td>Force an unmount or a remount change.</td>
</tr>
<tr>
<td>noatime</td>
<td>Set _MOUNT_NOATIME</td>
<td>Disable logging of file access times.</td>
</tr>
<tr>
<td>nocreat</td>
<td>Set _MOUNT_NOCREAT</td>
<td>Don’t allow file creation on the filesystem.</td>
</tr>
<tr>
<td>noexec</td>
<td>Set _MOUNT_NOEXEC</td>
<td>Don’t allow executables to load.</td>
</tr>
<tr>
<td>nostat</td>
<td>Set _MFLAG_OCB</td>
<td>Don’t attempt to <code>stat()</code> the special device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>before mounting (i.e. <code>-t</code>).</td>
</tr>
<tr>
<td>nosuid</td>
<td>Set _MOUNT_NOSUID</td>
<td>Don’t honor setuid bits on the filesystem.</td>
</tr>
<tr>
<td>opaque</td>
<td>Set _MOUNT_OPAQUE</td>
<td>Call <code>resmgr_attach()</code> with <code>RESMGR_FLAG_OPAQUE</code>.</td>
</tr>
<tr>
<td>remount</td>
<td>Set _MOUNT_REMOUNT</td>
<td>This path is already mounted; perform an update.</td>
</tr>
<tr>
<td>ro</td>
<td>Set _MOUNT_READONLY</td>
<td>Mark the filesystem mountpoint as read-only.</td>
</tr>
<tr>
<td>rw</td>
<td>Clear _MOUNT_READONLY</td>
<td>Mark the filesystem mountpoint as read/write.</td>
</tr>
<tr>
<td>suid</td>
<td>Clear _MOUNT_SUID</td>
<td>Honor setuid bits on the filesystem (default).</td>
</tr>
</tbody>
</table>

*continued*...
**mount_parse_generic_args()**  © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Option:</th>
<th>Set/Clear this bit:</th>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>Set _MOUNT_REMOUNT</td>
<td>This path is already mounted, perform an update.</td>
</tr>
</tbody>
</table>

**Returns:**

A string pointing to unprocessed options.

**Examples:**

```c
while ((c = getopt(argv, argo, "o:"))) {
    switch (c) {
    case 'o':
        if ((mysteryop = mount_parse_generic_args(optarg, &flags))) {
            /*
             * You can do your own getsubopt type processing here
             * mysteryop is stripped of the common options.
             */
        } break;
    }
}
```

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
See also:

`mount()`, `resmgr_attach()`, `umount()`

`mount` in the Utilities Reference

Writing a Resource Manager in Programmer’s Guide
Synopsis:

```c
#include <malloc.h>

enum mcheck_status mprobe(void * ptr);
```

Arguments:

- `ptr` A pointer to the start of the heap block.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mprobe()` function attempts to perform consistency checks on the allocated block specified by `ptr`, which was previously returned by a call to `calloc()`, `malloc()` or `realloc()`.

Consistency checks look for inconsistencies within the block header or in the block trailer byte. They may also detect block overruns.

The level of checking provided depends on which version of the allocator you’ve linked the application with:

- C library — minimal consistency checking.
- Nondebug version of the `malloc` library — a slightly greater level of consistency checking.
- Debug version of the `malloc` library — extensive consistency checking, with tuning available through the use of the `mallopt()` function.
mprobe()

Returns:

One of the values of the `mcheck_status` enumeration:

- **MCHECK_DISABLED**
  Consistency checking isn’t currently enabled, or consistency information isn’t available for this block.

- **MCHECK_OK**
  There are no inconsistencies in this block.

- **MCHECK_HEAD**
  The block header is corrupted.

- **MCHECK_TAIL**
  The block trailer byte is corrupted or there has been a block overrun.

- **MCHECK_FREE**
  The `ptr` argument doesn’t point to an allocated heap block.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Calling `mprobe()` on a pointer already deallocated by a call to `free()` or `realloc()` could corrupt the memory allocator’s data structures and result in undefined behavior.
See also:

`mallopt()`, `mcheck()`

The Heap Analysis chapter in the *Programmer’s Guide*. 
Synopsis:

```c
#include <sys/mman.h>

int mprotect( const void * addr,
              size_t len,
              int prot );
```

Arguments:

- `addr` The beginning of the range of addresses whose protection you want to change.
- `len` The length of the range of addresses, in bytes.
- `prot` The new access capabilities for the mapped memory region(s). You can combine the following bits, which are defined in `<sys/mman.h>`:
  - `PROT_EXEC` — the region can be executed.
  - `PROT_NOCACHE` — disable caching of the region (for example, to access dual ported memory).
  - `PROT_NONE` — the region can't be accessed.
  - `PROT_READ` — the region can be read.
  - `PROT_WRITE` — the region can be written.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mprotect()` function changes the access protections on any mappings residing in the range starting at `addr`, and continuing for `len` bytes.
mprotect()

Returns:

0    Success.
-1   An error occurred (errno is set).

If mprotect() fails, the protections on some of the pages in the address range starting at addr and continuing for len bytes may have been changed.

Errors:

EACCES    The memory object wasn’t opened for read, regardless of the protection specified.
           The memory object wasn’t opened for write, and PROT_WRITE was specified for a MAP_SHARED type mapping.
EAGAIN    The prot argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and there’s insufficient memory resources to reserve for locking the private pages (if required).
ENOMEM    The addresses in the range starting at addr and continuing for len bytes are outside the range allowed for the address space of a process, or specify one or more pages that are not mapped.
           The prot argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and locking the private pages (if required) would need more space than the system can supply to reserve for doing so.
ENOSYS    The function mprotect() isn’t supported by this implementation.
Classification:

POSIX 1003.1 MPR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`mmap()`, `munmap()`, `shm_open()`, `shm_unlink()`
mq_close()

Close a message queue

Synopsis:

```c
#include <mqueue.h>

int mq_close( mqd_t mqdes );
```

Arguments:

- `mqdes` The message-queue descriptor, returned by `mq_open()`, of the message queue that you want to close.

Library:

- For the traditional implementation, `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

- For the alternate implementation using asynchronous messages, `libmq`
  
  Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_close()` function removes the association between `mqdes` and a message queue. If the current process attaches a notify to this queue for notification, the attachment is eliminated. If this queue is unlinked before the call to `mq_close()`, and this process is the last process to call `mq_close()` on the queue, then the queue is destroyed, along with its contents.

Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the Utilities Reference.

In the traditional (mqueue) implementation, calling `close()` with `mqdes` has the same effect as calling `mq_close()`. 
mq_close()

Returns:
-1 if an error occurred (errno is set). Any other value indicates success.

Errors:
EBADF Invalid queue mqdes.

Classification:
POSIX 1003.1 MSG

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mq_open(), mq_unlink()

mq.mqueue in the Utilities Reference
**mq_getattr()**

Get a message queue’s attributes

**Synopsis:**

```c
#include <mqueue.h>

int mq_getattr( mqd_t mqdes,
                 struct mq_attr* mqstat );
```

**Arguments:**

- `mqdes` The message-queue descriptor, returned by `mq_open()`, of the message queue that you want to get the attributes of.
- `mqstat` A pointer to a `mq_attr` structure where the function can store the attributes of the message queue. For more information, see below.

**Library:**

- For the traditional implementation, `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

- For the alternate implementation using asynchronous messages, `libmq`
  
  Use the `-lmq` option to `qcc` to link against this library.

**Description:**

The `mq_getattr()` function determines the current attributes of the queue referenced by `mqdes`. These attributes are stored in the location pointed to by `mqstat`.

Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the *Utilities Reference*.

The fields of the `mq_attr` structure are as follows:
long `mq_flags`

The options set for this open message-queue description (i.e. these options are for the given `mqdes`, not the queue as a whole). This field may have been changed by call to `mq_setattr()` since you opened the queue.

- O_NONBLOCK — no call to `mq_receive()` or `mq_send()` will ever block on this queue. If the queue is in such a condition that the given operation can’t be performed without blocking, then an error is returned, and `errno` is set to EAGAIN.

long `mq_maxmsg`

The maximum number of messages that can be stored on the queue. This value was set when the queue was created.

long `mq_msgsize`

The maximum size of each message on the given message queue. This value was also set when the queue was created.

long `mq_curmsgs`

The number of messages currently on the given queue.

long `mq_sendwait`

The number of threads currently waiting to send a message. This field was eliminated from the POSIX standard after draft 9, but has been kept as a QNX Neutrino extension. A nonzero value in this field implies that the queue is full.

long `mq_recvwait`

The number of threads currently waiting to receive a message. Like `mq_sendwait`, this field has been kept as a QNX Neutrino extension. A nonzero value in this field implies that the queue is empty.
The alternate (mq) implementation of message queues doesn’t see the non-POSIX `mq_sendwait` and `mq_recvwait` fields.

**Returns:**

-1 if an error occurred (`errno` is set). Any other value indicates success.

**Errors:**

EBADF Invalid message queue `mqdes`.

**Classification:**

POSIX 1003.1 MSG

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

**See also:**

`mq_close()`, `mq_open()`, `mq_receive()`, `mq_send()`, `mq_setattr()`,

`mq`, `mqueue` in the *Utilities Reference*
mq_notify()
Ask to be notified when there’s a message in the queue

Synopsis:

```c
#include <mqueue.h>

int mq_notify(  
    mqd_t mqdes,  
    const struct sigevent* notification );
```

Arguments:

- `mqdes` The message-queue descriptor, returned by `mq_open()`, of the message queue that you want to get notification for.
- `notification` NULL, or a pointer to a `sigevent` structure that describes how you want to be notified.

Library:

- For the traditional implementation, `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
  Use the `-l mq` option to `qcc` to link against this library.

Description:

If `notification` isn’t NULL, the `mq_notify()` function asks the server to notify the calling process when the queue makes the transition from empty to nonempty. The means by which the server is to notify the process is passed in the `sigevent` structure pointed to by `notification`. Once the message queue server has notified the process of the transition, the notification is removed.
Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the *Utilities Reference*.

We recommend that you use the following event types in this case:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE

Under normal operation, only one process may register for notification at a time. If a process attempts to attach a notification, and another process is already attached, an error is returned and `errno` is set to EBUSY.

If a process has registered for notification, and another process is blocked on `mq_receive()`, then the `mq_receive()` call is satisfied by any arriving message. The resulting behavior is as if the message queue remained empty.

If `notification` is NULL and the current process is currently registered for notification, then the existing registration is removed.

**Returns:**

-1 if an error occurred (`errno` is set). Any other value indicates success.

**Errors:**

- EBADF  Invalid message queue `mqdes`.
- EBUSY   A process has already registered for notification for the given queue.
mq_notify()

Classification:

POSIX 1003.1 MSG

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mq_open(), mq_receive(), mq_send(), sigevent
mq.mqueue in the Utilities Reference
**Synopsis:**

```c
#include <mqueue.h>

mqd_t mq_open( const char * name, 
              int oflag, 
              ... )
```

**Arguments:**

- **name**
  The name of the message queue that you want to open; see below.

- **oflag**
  You must specify one of O_RDONLY (receive-only), O_WRONLY (send-only) or O_RDWR (send-receive). In addition, you can OR in the following constants to produce the following effects:
  
  - **O_CREAT** — if name doesn’t exist, instruct the server to create a new message queue with the given name. If you specify this flag, `mq_open()` uses its `mode` and `mq_attr` arguments; see below.
  
  - **O_EXCL** — if you set both O_EXCL and O_CREAT, and a message queue name exists, the call fails and `errno` is set to EEXIST. Otherwise, the queue is created normally. If you set O_EXCL without O_CREAT, it’s ignored.
  
  - **O_NONBLOCK** — under normal message queue operation, a call to `mq_send()` or `mq_receive()` could block if the message queue is full or empty. If you set this flag, these calls never block. If the queue isn’t in a condition to perform the given call, `errno` is set to EAGAIN and the call returns an error.

If you set O_CREAT in the `oflag` argument, you must also pass these arguments to `mq_open()`:

- **mode**
  The file permissions for the new queue. For more information, see “Access permissions” in the documentation for `stat()`.
If you set any bits other than file permission bits, they’re ignored. Read and write permissions are analogous to receive and send permissions; execute permissions are ignored.

*mq_attr* NULL, or a pointer to an *mq_attr* structure that contains the attributes that you want to use for the new queue. For more information, see *mq_getattr()*.

If *mq_attr* is NULL, the following default attributes are used — depending on which implementation of message queues you’re using — provided that you didn’t override the defaults when you started the message-queue server:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Traditional</th>
<th>Alternate</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mq_maxmsg</em></td>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td><em>mq_msgsize</em></td>
<td>4096</td>
<td>256</td>
</tr>
<tr>
<td><em>mq_flags</em></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If *mq_attr* isn’t NULL, the new queue adopts the *mq_maxmsg* and *mq_msgsize* of *mq_attr*. The *mq_flags* flags field is ignored.

**Library:**

- For the traditional implementation, *libc*
  Use the `-l libc` option to *qcc* to link against this library. This library is usually included automatically.

- For the alternate implementation using asynchronous messages, *libmq*
  Use the `-l mq` option to *qcc* to link against this library.
Description:

The \texttt{mq\_open()} function opens a message queue referred to by \texttt{name}, and returns a message queue descriptor by which the queue can be referenced in the future.

Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for \texttt{mq} and \texttt{mqueue} in the \textit{Utilities Reference}.

The name is interpreted as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>Pathname space entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>\texttt{cwd/entry}</td>
</tr>
<tr>
<td>/entry</td>
<td>\texttt{/dev/mqueue/entry} using the traditional \texttt{(mqueue)} implementation, or \texttt{/dev/mq/entry} using the alternate \texttt{(mq)} implementation</td>
</tr>
<tr>
<td>entry/newentry</td>
<td>\texttt{cwd/entry/newentry}</td>
</tr>
<tr>
<td>/entry/newentry</td>
<td>\texttt{/entry/newentry}</td>
</tr>
</tbody>
</table>

where \texttt{cwd} is the current working directory for the program at the point that it calls \texttt{mq\_open()}.

If you want to open a queue on another node, you have to use the traditional (\texttt{mqueue}) implementation and specify the name as \texttt{/net/node/mqueue\_location}.

If \texttt{name} doesn’t exist, \texttt{mq\_open()} examines the third and fourth parameters: a \texttt{mode\_t} and a pointer to an \texttt{mq\_attr} structure.

The only time that a call to \texttt{mq\_open()} with \texttt{O\_CREATE} set fails is if you open a message queue and later unlink it, but never close it. Like their file counterparts, an unlinked queue that hasn’t yet been closed must continue to exist; an attempt to recreate such a message queue fails, and \texttt{errno} is set to ENOENT.
Message queues persist — like files — even after the processes that created them end. A message queue is destroyed when the last process connected to it unlinks from the queue by calling `mq_unlink()`.

**Returns:**

A valid message queue descriptor if the queue is successfully created, or -1 (`errno` is set).

**Errors:**

- **EACCES** The message queue exists, and you don’t have permission to open the queue under the given `oflag`, or the message queue doesn’t exist, and you don’t have permission to create one.
- **EEXIST** You specified the `O_CREAT` and `O_EXCL` flags in `oflag`, and the queue `name` exists.
- **EINVAL** You specified the `O_CREAT` flag in `oflag`, and `mq_attr` wasn’t NULL, but some values in the `mq_attr` structure were invalid.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **EMFILE** Too many file descriptors are in use by the calling process.
- **ENAMETOOLONG** The length of `name` exceeds `PATH_MAX`.
- **ENFILE** Too many message queues are open in the system.
- **ENOENT** You didn’t set the `O_CREAT` flag, and the queue `name` doesn’t exist.
- **ENOSPC** The message queue server has run out of memory.
ENOSYS  The `mq_open()` function isn’t implemented for the filesystem specified in `name`.

**Classification:**

POSIX 1003.1 MSG

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`mq_close()`, `mq_getattr()`, `mq_notify()`, `mq_receive()`, `mq_send()`, `mq_setattr()`, `mq_timedreceive()`, `mq_timedsend()`, `mq_unlink()`

`mq`, `mqueue` in the *Utilities Reference*
mq_receive()

Receive a message from a queue

Synopsis:

```c
#include <mqueue.h>

ssize_t mq_receive( mqd_t mqdes,
                    char* msg_ptr,
                    size_t msg_len,
                    unsigned int* msg_prio );
```

Arguments:

- `mqdes` The message-queue descriptor, returned by `mq_open()`, of the message queue that you want to get a message from.
- `msg_ptr` A pointer to a buffer where the function can store the message received.
- `msg_len` The message size of the given queue.
- `msg_prio` NULL, or a pointer to a location where the function can store the priority of the message received.

Library:

- For the traditional implementation, `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
  Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_receive()` function is used to receive the oldest of the highest priority messages in the queue specified by `mqdes`. The priority of the message received is put in the location pointed to by `msg_prio`, the data itself in the location pointed to by `msg_ptr`, and the size received is be returned.
Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for \texttt{mq} and \texttt{mqueue} in the \textit{Utilities Reference}.

If you call \texttt{mq\_receive()} with a \texttt{msg\_len} of anything other than the \texttt{mq\_msgsize} of the specified queue, then \texttt{mq\_receive()} returns an error, and \texttt{errno} is set to \texttt{EINVAL}.

If there are no messages on the queue specified, and \texttt{O\_NONBLOCK} wasn’t set in \texttt{oflag} during \texttt{mq\_open()}, then the \texttt{mq\_receive()} call blocks. If multiple \texttt{mq\_receive()} calls are blocked on a single queue, then they’re unblocked in FIFO order as messages arrive.

In the traditional (\texttt{mqueue}) implementation, calling \texttt{read()} with \texttt{mqdes} is analogous to calling \texttt{mq\_receive()} with a NULL \texttt{msg\_prio}.

**Returns:**

The size of the message removed from the queue. If the call fails, -1 is returned as the size, no message is removed from the queue, and \texttt{errno} is set.

**Errors:**

\begin{itemize}
  \item \textbf{EAGAIN} The \texttt{O\_NONBLOCK} flag was set and there are no messages currently on the specified queue.
  \item \textbf{EBADF} The \texttt{mqdes} argument doesn’t represent a valid queue open for reading.
  \item \textbf{EINTR} The operation was interrupted by a signal.
  \item \textbf{EINVAL} The \texttt{msg\_ptr} argument isn’t a valid pointer, or \texttt{msg\_len} is less than 0, or \texttt{msg\_len} is less than the message size specified in \texttt{mq\_open()}. The default message size is 4096 bytes for the traditional (\texttt{mqueue}) implementation, and 256 bytes for the alternate (\texttt{mq}) implementation.
\end{itemize}
EMSGSIZE The given msg_len is shorter than the mq_msgsize for the given queue or the given msg_len is too short for the message that would have been received.

Classification:

POSIX 1003.1 MSG

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mq_close(), mq_open(), mq_send(), mq_timedreceive(), read()

mq.mqueue in the Utilities Reference
**mq_send()**

Send a message to a queue

**Synopsis:**
```
#include <mqueue.h>

int mq_send( mqd_t mqdes,  
             const char * msg_ptr,  
             size_t msg_len,  
             unsigned int msg_prio);
```

**Arguments:**
- `mqdes` The message-queue descriptor, returned by `mq_open()`, of the message queue that you want to send a message to.
- `msg_ptr` A pointer to the message that you want to send.
- `msg_len` The size of the message.
- `msg_prio` The priority of the message, in the range from 0 to (MQ_PRIO_MAX-1).

**Library:**
- For the traditional implementation, **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, **libmq**
  Use the `-l mq` option to `qcc` to link against this library.

**Description:**
The `mq_send()` function puts a message of size `msg_len` and pointed to by `msg_ptr` into the queue indicated by `mqdes`. The new message has a priority of `msg_prio`. 
Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the Utilities Reference.

The queue is maintained in priority order, and in FIFO order within the same priority.

If the number of elements on the specified queue is equal to its `mq_maxmsg`, and O_NONBLOCK (in oflag of `mq_open()`) has been set, the call to `mq_send()` blocks. It becomes unblocked when there’s room on the queue to send the given message. If more than one `mq_send()` is blocked on a given queue, and space becomes available in that queue to send, then the `mq_send()` with the highest priority message is unblocked.

In the traditional (`mqueue`) implementation, calling `write()` with `mqdes` is analogous to calling `mq_send()` with a `msg_prio` of 0.

**Returns:**

-1 if an error occurred (`errno` is set). Any other value indicates success.

**Errors:**

- **EAGAIN** The O_NONBLOCK flag was set when opening the queue, and the specified queue is full.
- **EBADF** The `mqdes` argument doesn’t represent a valid message queue descriptor, or `mqdes` wasn’t opened for writing.
- **EINTR** The call was interrupted by a signal.
- **EINVAL** One of the following cases is true:
  - `msg_len` was negative
  - `msg_prio` was greater than (MQ_PRIO_MAX-1)
mq_send()  

- msg_prio was less than 0

EMSGSIZE   The msg_len argument was greater than the msgsize associated with the specified queue.

Classification:

POSIX 1003.1 MSG

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mq_close(), mq_open(), mq_receive(), mq_timedsend()

mq, mqueue in the Utilities Reference
**mq_setattr()**

Set a queue’s attributes

### Synopsis:

```c
#include <mqueue.h>

int mq_setattr( mqd_t mqdes,
                const struct mq_attr* mqstat,
                struct mq_attr* omqstat );
```

### Arguments:

- **mqdes**: The message-queue descriptor, returned by `mq_open()`, of the message queue that you want to set the attributes of.
- **mqstat**: A pointer to a `mq_attr` structure that specifies the attributes that you want to use for the message queue. For more information about this structure, see `mq_getattr()`; for information about which attributes you can set, see below.
- **omqstat**: NULL, or a pointer to a `mq_attr` structure where the function can store the old attributes of the message queue.

### Library:

- For the traditional implementation, **libc**
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

- For the alternate implementation using asynchronous messages, **libmq**
  
  Use the `-l mq` option to `qcc` to link against this library.

### Description:

The `mq_setattr()` function sets the `mq_flags` field for the specified queue (passed as the `mq_flags` field in `mqstat`). If `omqstat` isn’t NULL, then the old attribute structure is stored in the location that it points to.
Neutrino supports two implementations of message queues: a
traditional implementation, and an alternate one that uses
asynchronous messages. For more information, see the entry for `mq`
and `mqueue` in the Utilities Reference.

This function ignores the `mq_maxmsg`, `mq_msgsize`, and
`mq_curmsgs` fields of `mqstat`. The `mq_flags` field is the bitwise OR of
zero or more of the following constants:

- **O_NONBLOCK**
  No `mq_receive()` or `mq_send()` will ever block on this queue. If
  the queue is in such a condition that the given operation can’t be
  performed without blocking, then an error is returned, and
  `errno` is set to EAGAIN.

The settings that you make for `mq_flags` apply only to the given
message-queue description (i.e. locally), not to the queue itself.

**Returns:**

-1 if the function couldn’t change the attributes (`errno` is set). Any
other value indicates success.

**Errors:**

- **EBADF**  Invalid message queue `mqdes`.

**Classification:**

POSIX 1003.1 MSG

**Safety**

Cancellation point  No

continued...
Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mq_setattr(), mq_open(), mq_receive(), mq_send()

mq, mqueue in the Utilities Reference
Receive a message from a message queue

Synopsis:
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(
    mqd_t mqdes,
    char * msg_ptr,
    size_t msg_len,
    unsigned int * msg_prio,
    const struct timespec * abs_timeout);

Arguments:
- `mqdes` The descriptor of the message queue you want to receive a message from, returned by `mq_open()`.
- `msg_ptr` A pointer to a buffer where the function can store the message data.
- `msg_len` The size of the buffer, in bytes.
- `msg_prio` NULL, or a pointer to a location where the function can store the priority of the message that it removed from the queue.
- `abs_timeout` A pointer to a `timespec` structure that specifies the `absolute` time (not the relative time to the current time) to wait before the function stops trying to receive messages.

Library:
- For the traditional implementation, `libc`
  Use the `--l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
  Use the `--l mq` option to `qcc` to link against this library.
**mq_timedreceive()**

**Description:**

The `mq_timedreceive()` function receives the oldest of the highest priority messages in the queue specified by `mqdes`.

Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the Utilities Reference.

If you call `mq_timedreceive()` with a `msg_len` of anything other than the `mq_msgsize` of the specified queue, then `mq_timedreceive()` returns an error, and `errno` is set to EINVVAL.

If there are no messages on the queue specified, and O_NONBLOCK isn’t set in `oflag` during `mq_open()`, then the `mq_timedreceive()` call blocks. If multiple `mq_timedreceive()` calls are blocked on a single queue, then they’re unblocked in FIFO order as messages arrive.

In the traditional (`mqueue`) implementation, calling `read()` with `mqdes` is analogous to calling `mq_timedreceive()` with a NULL `msg_prio`.

**Returns:**

The size of the message removed from the queue, or -1 if an error occurred (no message is removed from the queue, and `errno` is set).

**Errors:**

- **EAGAIN** The O_NONBLOCK flag was set and there are no messages currently on the specified queue.
- **EBADF** The `mqdes` argument doesn’t represent a valid queue open for reading.
- **EINTR** The operation was interrupted by a signal.
- **EINVAL** The `msg_ptr` argument isn’t a valid pointer, or `msg_len` is less than 0, or `msg_len` is less than the message size specified in `mq_open()`. The default message size is 4096 bytes.
EMSGSIZE  The given msg_len is shorter than the mq_msgsize for the given queue or the given msg_len is too short for the message that would have been received.

ETIMEDOUT The timeout value was exceeded.

Examples:

Specify an absolute timeout of 1 second:

```c
struct timespec tm;

clock_gettime(CLOCK_REALTIME, &tm);
.tm.tv_sec += 1;
if( 0 > mq_timedreceive( fd, buf, 4096, NULL, t ) ) {
    ...
}
```

Classification:

POSIX 1003.1 MSG

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

mq_close(), mq_open(), mq_receive(), mq_send(), mq_timedsend(), timespec

mq, mqueue in the Utilities Reference
mq_timedsend()

Send a message to a message queue

Synopsis:

```c
#include <mqueue.h>
#include <time.h>

int mq_timedsend( mqd_t mqdes,
                 const char * msg_ptr,
                 size_t msg_len,
                 unsigned int msg_prio,
                 const struct timespec * abs_timeout );
```

Arguments:

- `mqdes`: The descriptor of the message queue you want to put the message into, returned by `mq_open()`.
- `msg_ptr`: A pointer to the message data.
- `msg_len`: The size of the buffer, in bytes.
- `msg_prio`: The priority of the message, in the range from 0 to (MQ_Prio_MAX-1).
- `abs_timeout`: A pointer to a `timespec` structure that specifies the `absolute` time (not the relative time to the current time) to wait before the function stops trying to receive messages.

Library:

- For the traditional implementation, `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
- For the alternate implementation using asynchronous messages, `libmq`
  Use the `-l mq` option to `qcc` to link against this library.
**mq_timedsend()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The `mq_timedsend()` function puts a message of size `msg_len` and pointed to by `msg_ptr` into the queue indicated by `mqdes`. The new message has a priority of `msg_prio`.

Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the *Utilities Reference*.

The queue maintained is in priority order, and in FIFO order within the same priority.

If the number of elements on the specified queue is equal to its `mq_maxmsg`, and `O_NONBLOCK` (in `oflag` of `mq_open()`) hasn’t been set, the call to `mq_timedsend()` blocks. It becomes unblocked when there’s room on the queue to send the given message. If more than one `mq_timedsend()` is blocked on a given queue, and space becomes available in that queue to send, then the `mq_timedsend()` with the highest priority message is unblocked.

In the traditional (`mqueue`) implementation, calling `write()` with `mqdes` is analogous to calling `mq_timedsend()` with a `msg_prio` of 0.

**Returns:**

-1 if an error occurred (*errno* is set). Any other value indicates success.

**Errors:**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The O_NONBLOCK flag was set when opening the queue, and the specified queue is full.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The <code>mqdes</code> argument doesn’t represent a valid message queue descriptor, or <code>mqdes</code> isn’t opened for writing.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The call was interrupted by a signal.</td>
</tr>
</tbody>
</table>
mq_timedsend()

EINVAL One of the following is true:
- The \texttt{msg\_len} is negative.
- The \texttt{msg\_prio} is greater than (MQ\_PRIO\_MAX-1).
- The \texttt{msg\_prio} is less than 0.
- The MQ\_PRIO\_RESTRICT flag is set in the \texttt{mq\_attr} member of \texttt{mqdes}, and \texttt{msg\_prio} is greater than the priority of the calling process.

EMSGSIZE The \texttt{msg\_len} argument is greater than the \texttt{msgsize} associated with the specified queue.

ETIMEDOUT The timeout value was exceeded.

Examples:
See the example for \texttt{mq\_timedreceive()}.

Classification:

POSIX 1003.1 MSG

Safety

\begin{tabular}{|c|c|}
\hline
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}

See also:

\texttt{mq\_close()}, \texttt{mq\_open()}, \texttt{mq\_receive()}, \texttt{mq\_send()}, \texttt{mq\_timedreceive()}, \texttt{timespec}

\texttt{mq, mqueue} in the \textit{Utilities Reference}
mq_unlink()

Remove a queue

Synopsis:

```
#include <mqueue.h>

int mq_unlink( const char* name );
```

Arguments:

- `name` The name of the message queue that you want to unlink.

Library:

- For the traditional implementation, `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

- For the alternate implementation using asynchronous messages, `libmq`
  
  Use the `-l mq` option to `qcc` to link against this library.

Description:

The `mq_unlink()` function removes the queue with the given `name`.

Neutrino supports two implementations of message queues: a traditional implementation, and an alternate one that uses asynchronous messages. For more information, see the entry for `mq` and `mqueue` in the Utilities Reference.

If some process has the queue open when the call to `mq_unlink()` is made, then the actual deletion of the queue is postponed until it has been closed. If a queue exists in the netherworld between unlinking and the actual removal of the queue, then all calls to open a queue with the given name fail (even if `O_CREAT` is present in `oflag`). Once the queue is deleted, all elements currently on it are freed. Due to the lazy deletion of queues, it’s impossible for any process to be blocked on the message queue when it’s deleted.
Calling *unlink()* with a name that resolves to the message queue server’s namespace (e.g. /dev/mqueue/my_queue) is analogous to calling *mq_unlink()* with *name* set to the last elements of the pathname (e.g. my_queue).

**Returns:**

-1 if the queue wasn’t successfully unlinked (*errno* is set). Any other value indicates that the queue was successfully unlinked.

**Errors:**

- **EACCES** You don’t have permission to unlink the specified queue.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG** The length of *name* exceeds PATH_MAX.
- **ENOENT** The queue *name* doesn’t exist.
- **ENOSYS** The *mq_unlink()* function isn’t implemented for the filesystem specified in *path*.

**Classification:**

POSIX 1003.1 MSG

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

mq_close(), mq_open(), unlink()

mq, mqueue in the Utilities Reference
Synopsis:

```c
#include <stdlib.h>

long mrand48( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `mrand48()` function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a signed `long` integer uniformly distributed over the interval \([-2^{31}, 2^{31})\).

Call one of `lcong48()`, `seed48()`, or `srand48()` to initialize the random-number generator before calling `drand48()`, `lrand48()`, or `mrand48()`.

The `jrand48()` function is a thread-safe version of `mrand48()`.

Returns:

A pseudo-random `long` integer.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

`drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `nrand48()`, `seed48()`, `srand48()`
Synopsis:

```c
struct _msg_info {       /* _msg_info _server_info */
    uint32_t  nd;       /* client server */
    uint32_t  srcnd;    /* server n/a */
    pid_t    pid;       /* client server */
    int32_t  tid;       /* thread n/a */
    int32_t  chid;      /* server server */
    int32_t  scoid;     /* server server */
    int32_t  coid;      /* client client */
    int32_t  msglen;    /* msg n/a */
    int32_t  srcmsglen; /* thread n/a */
    int32_t  dstmsglen; /* thread n/a */
    int16_t  priority;  /* thread n/a */
    int16_t  flags;     /* n/a client */
    uint32_t reserved;
};
```

Description:

The `_msg_info` structure contains information about a message. The members include:

- **nd**: The node descriptor of the client machine as viewed by the server. See "Node descriptors," below.
- **srcnd**: The node descriptor of the server, as viewed by the client.
- **pid**: The process ID of the sending thread.
- **tid**: The thread ID of the sending thread.
- **chid**: The channel ID that the message was received on.
- **scoid**: The server connection ID.
- **coid**: The client connection ID.
- **msglen**: The number of bytes received.
srcmsglen  The length of the source message, in bytes, as sent by
MsgSend*(). This may be greater than the value in
msglen. This member is valid only if you set
_NTO_CHF_SENDER_LEN in the flags argument to
ChannelCreate() for the channel that received the
message.

dstmsglen  The length of the client’s reply buffer, in bytes, as
passed to MsgSend*(). This member is valid only if
you set _NTO_CHF_REPLY_LEN in the flags argument
to ChannelCreate() for the channel that received the
message.

priority  The priority of the sending thread.

flags  The client has an unblock pending
_NTO_MI_UNBLOCK_REQ (i.e. a timeout on the send
occurred or a signal was delivered and
_NTO_CHF_UNBLOCK is set on the channel).

---

The msglen and srcmsglen members are valid only until the next call
to MsgRead*() or MsgWrite*().

If msglen is less than srcmsglen and is also less than the receive buffer
size, the message is a network transaction that requires more reading
of data with MsgRead*().

Node descriptors

The nd (node descriptor) is a temporary numeric description of a
remote node. For more information, see the Qnet Networking chapter
of the System Architecture guide.
<table>
<thead>
<tr>
<th>To:</th>
<th>Use this function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare two <em>nd</em> objects</td>
<td>ND_NODE_CMP()</td>
</tr>
<tr>
<td>Convert a <em>nd</em> to text</td>
<td>netmgr_ndtostr()</td>
</tr>
<tr>
<td>Convert text to a <em>nd</em></td>
<td>netmgr_strtond()</td>
</tr>
</tbody>
</table>

**Classification:**
QNX Neutrino

**See also:**
MsgInfo(), MsgReceive(), MsgReceivev(), ND_NODE_CMP(), netmgr_ndtostr(), netmgr_strtond()
Deliver an event through a channel

Synopsis:

```c
#include <sys/neutrino.h>

int MsgDeliverEvent( int rcvid,
                      const struct sigevent* event );

int MsgDeliverEvent_r( int rcvid,
                        const struct sigevent* event );
```

Arguments:

- `rcvid` The value returned to the server when it receives a message from a client using `MsgReceive*()`.
- `event` A pointer to a `sigevent` structure that contains the event you want to send. These events are defined in `<sys/siginfo.h>`. The type of event is placed in `event.sigev_notify`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgDeliverEvent()` and `MsgDeliverEvent_r()` kernel calls deliver an `event` from a server to a client through a channel connection. They’re typically used to perform async IO and async event notification to clients that don’t want to block on a server.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Although the server can explicitly send any event it desires, it’s more typical for the server to receive a `struct sigevent` in a message from the client that already contains this data. The message also contains information for the server indicating the conditions on when
to notify the client with the event. The server then saves the rcvid from `MsgReceive*()` and the event from the message without needing to examine the event in any way. When the trigger conditions are met in the server, such as data becoming available, the server calls `MsgDeliverEvent()` with the saved rcvid and event.

You can use the SIGEV_SIGNAL set of notifications to create an asynchronous design in which the client is interrupted when the event occurs. The client can make this synchronous by using the `SignalWaitinfo()` kernel call to wait for the signal. Where possible, you should use an event-driven synchronous design that’s based on SIGEV_PULSE. In this case, the client sends messages to servers, and requests event notification via a pulse.

You’re not likely to use the event types SIGEV_UNBLOCK and SIGEV_INTR with this call.

You should use `MsgDeliverEvent()` when two processes need to communicate with each other without the possibility of deadlock. The blocking nature of `MsgSend*()` introduces a hierarchy of processes in which “sends” flow one way and “replies” the other way.

In the following diagram, processes at the A level can send to processes at the B or C level. Processes at the B level can send to the C level but they should never send to the A level. Likewise, processes at the C level can never send to those at the A or B level. To A, B and C are servers. To B, A is a client and C is a server.

A hierarchy of processes.

These hierarchies are simple to establish and ensure a clean
deadlock-free design. If these rules are broken then deadlock can occur as shown below:

```
A sends to "B"
B sends to "C"
C sends to "A"
```

Deadlock

A deadlock when sending messages improperly among processes.

There are common situations which require communication to flow backwards through the hierarchy. For example, A sends to B requesting notification when data is available. B immediately replies to A. At some point in the future, B will have the data A requested and will inform A. B can't send a message to A because this might result in deadlock if A decided to send to B at the same time.

The solution is to have B use a nonblocking `MsgDeliverEvent()` to inform A. A receives this pulse and sends a message to B requesting the data. B then replies with the data. This is the basis for asynchronous IO. Clients send to servers and where necessary, servers use pulses to request clients to resend to them as needed. This is illustrated below:

<table>
<thead>
<tr>
<th>Message</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sends to → B</td>
<td>Async IO request</td>
</tr>
<tr>
<td>B replies to → A</td>
<td>Request acknowledged</td>
</tr>
</tbody>
</table>

continued...
Message | Use
--- | ---
B sends pulse to → A | Requested data available
A sends to → B | Request for the data
B replies to → A | Reply with data

In client/server designs, you typically use *MsgDeliverEvent()* in the server, and *MsgSendPulse()* in the client.

**Blocking states**

None. In the network case, lower priority threads may run.

**Native networking**

When you use *MsgDeliverEvent()* to communicate across a network, the return code isn’t “reliable”. In the local case, *MsgDeliverEvent()* always returns a correct success or failure value. But since *MsgDeliverEvent()* must be nonblocking, in the networked case, the return value isn’t guaranteed to reflect the actual result on the client’s node. This is because *MsgDeliverEvent()* would have to block waiting for the communications between the two npm-qnets.

Generally, this isn’t a problem, because *MsgDeliverEvent()* is for the benefit of the client anyway — if the client no longer exists, then the client obviously doesn’t care that it didn’t get the event. The server usually delivers the event and then goes about its business, regardless of the success or failure of the event delivery.

**Returns:**

The only difference between these functions is the way they indicate errors:

*MsgDeliverEvent()*

If an error occurs, -1 is returned and *errno* is set. Any other value returned indicates success.
**MsgDeliverEvent()**, **MsgDeliverEvent_r()** © 2007, QNX Software Systems GmbH & Co. KG.

**MsgDeliverEvent_r()**

EOK is returned on success. This function does **NOT** set **errno**. If an error occurs, any value in the Errors section may be returned.

**Errors:**

- **EAGAIN** The kernel has insufficient resources to enqueue the event.
- **EBADF** The thread indicated by *rcvid* had its connection detached.
- **EFAULT** A fault occurred when the kernel tried to access the buffers provided.
- **ESRCH** The thread indicated by *rcvid* doesn’t exist.
- **ESRVRFAULT** A fault occurred in the server’s address space when it tried to write the pulse message to the server’s receive message buffer (SIGEV_PULSE only).

**Examples:**

The following example demonstrates how a client can request a server to notify it with a pulse at a later time (in this case, after the server has slept for two seconds). The server side notifies the client using **MsgDeliverEvent()**.

Here’s the header file that’s used by *client.c* and *server.c*:

```c
struct my_msg
{
    short type;
    struct sigevent event;
};
```

```c
#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL+5
#define MSG_GIVE_PULSE _IO_MAX+4
#define MY SERV "my_server_name"
```
Here's the client side that fills in a `struct sigevent` and then receives a pulse:

```c
/* client.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include "my_hdr.h"

#include "my_hdr.h"

int main( int argc, char **argv)
{
    int chid, coid, srv_coid, rcvid;
    struct my_msg msg;
    struct _pulse pulse;

    /* we need a channel to receive the pulse notification on */
    chid = ChannelCreate( 0 );

    /* and we need a connection to that channel for the pulse to be 
       delivered on */
    coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );

    /* fill in the event structure for a pulse */
    SIGEV_PULSE_INIT( &msg.event, coid, SIGEV_PULSE_PRIO_INHERIT,
                      MY_PULSE_CODE, 0 );
    msg.type = MSG_GIVE_PULSE;

    /* find the server */
    if ( (srv_coid = name_open( MY_SERV, 0 )) == -1)
    {
        printf("failed to find server, errno %d\n", errno);
        exit(1);
    }

    /* give the pulse event we initialized above to the server for 
       later delivery */
    MsgSend( srv_coid, &msg, sizeof(msg), NULL, 0 );

    /* wait for the pulse from the server */
    rcvid = MsgReceivePulse( chid, &pulse, sizeof(pulse), NULL );
    printf("got pulse with code %d, waiting for %d\n", pulse.code,
           MY_PULSE_CODE );

    return 0;
}
```
Here’s the server side that delivers the pulse defined by the `struct sigevent`:

```c
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include "my_hdr.h"

int main( int argc, char **argv) {
    int rcvid;
    struct my_msg msg;
    name_attach_t *attach;

    /* attach the name the client will use to find us */
    /* our channel will be in the attach structure */
    if ( (attach = name_attach( NULL, MY_SERV, 0 )) == NULL) {
        printf("server:failed to attach name, errno %d\n", errno );
        exit(1);
    }

    /* wait for the message from the client */
    rcvid = MsgReceive( attach->chid, &msg, sizeof( msg ), NULL );
    MsgReply(rcvid, 0, NULL, 0);
    if ( msg.type == MSG_GIVE_PULSE ) {
        /* wait until it is time to notify the client */
        sleep(2);

        /* deliver notification to client that client requested */
        MsgDeliverEvent( rcvid, &msg.event );
        printf("server:delivered event\n";
    } else {
        printf("server: unexpected message \n";
    }

    return 0;
}
```
MsgDeliverEvent(),
MsgDeliverEvent_r()

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

In the case of a pulse event, if the server faults on delivery, the pulse is either lost or an error is returned.

See also:

MsgReceive(), MsgReceivev(), MsgSend(), MsgSendPulse(),
MsgSendv(), sigevent, SignalWaitinfo()
**MsgError(), MsgError_r()**

Unblock a client and set its errno

**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgError( int rcvid,
              int error );

int MsgError_r( int rcvid,
                int error );
```

**Arguments:**

- `rcvid` The receive ID that `MsgReceive*()` returned.
- `error` The error code that you want to set for the client.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgError()` and `MsgError_r()` kernel calls unblock the client’s `MsgSend*()` call and set the client’s `errno` to `error`. No data is transferred.

If `error` is EOK, the `MsgSend*()` call returns EOK; if `error` is any other value, the `MsgSend*()` call returns -1.

These functions are identical except in the way they indicate errors. See the Returns section for details.

An error number of ERESTART causes the sender to immediately call `MsgSend*()` again. Since send and reply buffers passed to `MsgSend()` may overlap, you shouldn’t use ERESTART after a call to `MsgWrite()`.
MsgError(), MsgError_r()

Blocking states

None. In the network case, lower priority threads may run.

Native networking

MsgError() has increased latency when you use it to communicate across a network — the server is now writing the error code to its local npm-qnet, which may need to communicate with the client’s npm-qnet to actually transfer the error code.

Returns:

The only difference between these functions is the way they indicate errors:

MsgError() If an error occurs, the function returns -1 and and sets errno. Any other value returned indicates success.

MsgError_r() Returns EOK on success. This function does NOT set errno. If an error occurs, the function returns one of the values listed in the Errors section.

Errors:

ESRCH The thread indicated by rcvid doesn’t exist.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
See also:

ChannelCreate(), MsgRead(), MsgReadv(), MsgReceive(),
MsgReceivev(), MsgSend(), MsgSendv()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgInfo(int rcvid,  
              struct _msg_info* info);

int MsgInfo_r(int rcvid,  
               struct _msg_info* info);
```

Arguments:

- `rcvid` The return value from `MsgReceive*()`.
- `info` A pointer to a `_msg_info` structure where the function can store information about the message.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgInfo()` and `MsgInfo_r()` kernel calls get additional information about a received message and store it in the specified `_msg_info` structure.

These functions are identical, except in the way they indicate errors. See the Returns section for details.

The `info->msglen` and `info->srcmsglen` members are valid only until the next call to `MsgRead*()` or `MsgWrite*()`.

Blocking states

This call doesn’t block.
Returns:

The only difference between these functions is the way they indicate errors:

(MsgInfo())
If an error occurs, -1 is returned and errno is set. Any other value returned indicates success.

(MsgInfo_r())
EOK is returned on success. This function does NOT set errno. If an error occurs, any value in the Errors section may be returned.

Errors:

EFAULT A fault occurred when the kernel tried to access the buffers provided.
ESRCH The thread indicated by rcvid doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

ChannelCreate(), _msg_info, MsgRead(), MsgReadv(), MsgReceive(), MsgReceivev(), MsgSend(), MsgSendv(),
Synopsis:

```c
#include <sys/neutrino.h>

int MsgKeyData( int rcvid,  
                int op,  
                uint32_t key,  
                uint32_t * key2,  
                const iov_t * msg,  
                int parts );

int MsgKeyData_r( int rcvid,  
                  int op,  
                  uint32_t key,  
                  uint32_t * key2,  
                  const iov_t * msg,  
                  int parts );
```

Arguments:

- `rcvid` The return value from `MsgReceive*()`.
- `op` The operation to perform; one of:
  - `_NTO_KEYDATA_CALCULATE` — calculate a new key.
  - `_NTO_KEYDATA_VERIFY` — verify the key.
- `key` A private value for key (this can be a value returned by the `rand()` function).
- `key2` A pointer to a key. What the function stores in this location depends on the `op` argument:
  - `_NTO_KEYDATA_CALCULATE` — the new key.
  - `_NTO_KEYDATA_VERIFY` — zero if no tampering has occurred.
- `msg` A pointer to a portion of the reply data to be keyed.
- `parts` The number of parts in `msg`. 
**Message Key Data (MsgKeyData(), MsgKeyData_r())**

**Library:**

```
libc
```

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgKeyData()` and `MsgKeyData_r()` kernel calls allow two privileged processes to pass data through a common client while verifying that the client hasn’t modified the data. This is best explained by an example.

These functions are identical except in the way they indicate errors. See the Returns section for details.

A program calls `open()` with a filename. The `open()` function sends a message to the Process Manager, which is responsible for pathname management.

```
Client          MsgSendv open "filename"    Process Manager
```

*`MsgSendv()`, client to process manager.*

The Process Manager resolves the pathname, resulting in a fully qualified network path and the process ID to send the `open()` request to. This information is replied to the client.

```
Client         MsgReplyv pid, "/net/dan/filename"        Process Manager
```

*`MsgReplyv()`, process manager to client.*

The client now sends this message to `pid` with the fully qualified pathname.
MsgSendv(), client to filesystem manager

Note that the client can change the pathname before it sends it to the Filesystem Manager. In fact, it could skip the call to the Process Manager and manufacture any pathname it desired. The Filesystem Manager always performs permission checking. Therefore, changing or manufacturing pathnames isn’t normally something to be concerned about, except in one case: chroot() lets you specify a prefix that must be applied to all pathnames.

In the above example, the client may have had a chroot() of /net/node2/home/dan. This should limit the process from accessing files outside of /net/node2/home/dan. For example:

<table>
<thead>
<tr>
<th>User path</th>
<th>Mapped to chroot() path</th>
</tr>
</thead>
<tbody>
<tr>
<td>/bin/ls</td>
<td>/net/node2/home/dan/bin/ls</td>
</tr>
<tr>
<td>/</td>
<td>/net/node2/home/dan</td>
</tr>
</tbody>
</table>

The process has had its root set to a subdirectory, limiting the files it can access. For this to work, it’s necessary to prevent the client from changing or manufacturing its own pathnames.

In QNX Neutrino, only the Process Manager handles a user chroot(). Unlike a monolithic kernel where the filesystem shares the same address space as the kernel and the chroot() information, QNX I/O managers reside in separate address spaces and might not even reside on the same machine.

The solution to this problem is the MsgKeyData() call. When the Process Manager receives the open() message, it generates the reply data. Before replying, it calls MsgKeyData(), with these arguments:
MsgKeyData(), MsgKeyData_r() © 2007, QNX Software Systems GmbH & Co. KG.

rcvid  The return value from MsgReceive*().

op     _NTO_KEYDATA_CALCULATE

key    A private value for key (this can be a value returned by the rand() function).

key2   A pointer to a new key that should be returned to the client in a unkeyed area of the message.

msg    A pointer to a portion of the reply data to be keyed.

parts  The number of parts in msg.

The client now sends the message to the File Manager. On receipt of the message, the File Manager calls MsgKeyData() with the same arguments as above, except for:

op     _NTO_KEYDATA_VERIFY

key    The key that’s provided in the message.

MsgKeyData() sets the key pointed to by key2 to zero if no tampering has occurred.

Note that there are actually two keys involved. A public key that’s returned to the client and a private key that the Process Manager generated. The algorithm uses both keys and the data for verification.

Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

MsgKeyData() If an error occurs, -1 is returned and errno is set. Any other value returned indicates success.
MsgKeyData_r()  EOK is returned on success. This function does NOT set errno. If an error occurs, any value in the Errors section may be returned.

Errors:

ESRCH    The thread indicated by rcvid doesn’t exist.

EFAULT   A fault occurred when the kernel tried to access the buffers provided.

Examples:

/*
 * This program demonstrates the use of MsgKeyData() as a way
 * of a client handing off data from a source server to a
 * destination server such that if the client tampers with
 * the data, the destination server will know about it.
 */

#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/neutrino.h>

typedef struct {
    int    public_key;
    char   text[10];
} IPC_t;

int chid_src, chid_dst;

void* server_src_thread(void* parm);
void* server_dst_thread(void* parm);

main()
{
    pthread_t    tid[2];
    IPC_t        msg;
    int          coid;
    int          status;

    pthread_create(&tid[0], NULL, server_src_thread, NULL);

    server_dst_thread(NULL);
}
pthread_create(&tid[1], NULL, server_dst_thread, NULL);
sleep(3);
/* give time for channels to be created, sloppy but simple */

/*
 * Send to server_src_thread for some data.
 * The data will include some text and a public
 * key for that text.
 */
coid = ConnectAttach(0, 0, chid_src, 0, 0);
MsgSend(coid, NULL, 0, &msg, sizeof(msg));
ConnectDetach(coid);

/*
 * Now send to server_dst_thread with the reply from
 * server_src_thread. We didn’t modify the ‘text’ so it
 * should reply success. Note that we’re including the
 * public key.
 */
coid = ConnectAttach(0, 0, chid_dst, 0, 0);
status = MsgSend(coid, &msg, sizeof(msg), &msg, sizeof(msg));
printf("Sent unmodified text to server_dst_thread.
       Replied with %s\n", status == EOK ? "EOK" : "EINVAL" );

/*
 * Now tamper with the original ‘text’ (which we aren’t
 * supposed to do) and send to server_dst_thread again
 * but with the modified ‘text’ and the public key.
 * Since we tampered with the ‘text’, server_dst_thread
 * should reply failure.
 */
strcpy(msg.text, "NEWDATA");
status = MsgSend(coid, &msg, sizeof(msg), &msg, sizeof(msg));
printf("Sent modified text to server_dst_thread.
       Replied with %s\n", status == EOK ? "EOK" : "EINVAL" );

return 0;

void* server_src_thread(void* parm)
{
    int rcvid;
    int private_key; /* the kernel keeps this */
    iov_t keyed_area iov;
    IPC_t msg;
    struct timespec t;
}
chid_src = ChannelCreate(0);
while (1) {
    rcvid = MsgReceive(chid_src, &msg, sizeof(msg), NULL);

    /*
    * Give MsgKeyData() the private key and it will
    * calculate a public key for the 'text' member of
    * the message. The kernel will keep the private key
    * and we reply with the public key.
    * Note that we use the number of nanoseconds since the
    * last second as a way of getting a 32-bit pseudo
    * random number for the private key.
    */
    clock_gettime(CLOCK_REALTIME, &t);
    private_key = t.tv_nsec; /* nanoseconds since last second */
    STRCPY(msg.text, "OKDATA");
    SETIOV(&keyed_area_iov, &msg.text, sizeof(msg.text));
    MsgKeyData(rcvid, _NTO_KEYDATA_CALCULATE, private_key, &msg.public_key, &keyed_area_iov, 1);
    MsgReply(rcvid, 0, &msg, sizeof(msg));
}
return NULL;
}

void* server_dst_thread(void* parm)
{
    int rcvid, tampered, status;
    iov_t keyed_area_iov;
    IPC_t msg;

    chid_dst = ChannelCreate(0);
    while (1) {
        rcvid = MsgReceive(chid_dst, &msg, sizeof(msg), NULL);

        /*
        * Use the public key to see if the data
        * has been tampered with.
        */
        SETIOV(&keyed_area_iov, &msg.text, sizeof(msg.text));
        MsgKeyData(rcvid, _NTO_KEYDATA_VERIFY, msg.public_key, &tampered, &keyed_area_iov, 1);
        if (tampered)
            status = EINVAL; /* reply: 'text' was modified */
        else
            status = EOK; /* reply: 'text' was okay */
        MsgReply(rcvid, status, &msg, sizeof(msg));
    }
}

September 10, 2007
MsgKeyData(), MsgKeyData_r()

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chroot(), MsgReceive(), MsgReceivev(), open(), rand()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgRead(int rcvid,
    void* msg,
    int bytes,
    int offset);

int MsgRead_r(int rcvid,
    void* msg,
    int bytes,
    int offset);
```

Arguments:

- `rcvid` : The value returned by `MsgReceive*()` when you received the message.
- `msg` : A pointer to a buffer where the function can store the data.
- `bytes` : The number of bytes that you want to read. These functions don't let you read past the end of the thread's message; they return the number of bytes actually read.
- `offset` : An offset into the thread's send message that indicates where you want to start reading the data.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgRead()` and `MsgRead_r()` kernel calls read data from a message sent by a thread identified by `rcvid`. The thread being read from must not have been replied to and will be in the REPLY-blocked state. Any thread in the receiving process is free to read the message.
These functions are identical except in the way they indicate errors. See the Returns section for details.

The data transfer occurs immediately and the thread doesn’t block. The state of the sending thread doesn’t change.

You’ll use these functions in these situations:

- A message is sent consisting of a fixed header and a variable amount of data. The header contains the byte count of the data. If the data is large and has to be inserted into one or more buffers (like a filesystem cache), rather than read the data into one large buffer and then copy it into several other buffers, `MsgReceive()` reads only the header, and you can call `MsgRead()` one or more times to read data directly into the required buffer(s).

- A message is received but can’t be handled at the present time. At some point in the future, an event will occur that will allow the message to be processed. Rather than saving the message until it can be processed (thus using memory resources), you can use `MsgRead()` to reread the message, during which time the sending thread is still blocked.

- Messages that are larger than available buffer space are received. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use `MsgRead()` to read the message in small pieces, and use `MsgWrite*()` to write the messages in small pieces.

When you’re finished using `MsgRead()`, you must use `MsgReply*()` to ready the REPLY-blocked process and complete the message exchange.

**Blocking states**

None. In the network case, lower priority threads may run.
Native networking

The `MsgRead()` function has increased latency when it’s used to communicate across a network — a message pass is involved from the server to the network manager (at least). Depending on the size of the data transfer, the server’s `npm-qnet` and the client’s `npm-qnet` may need to communicate over the link to read more data bytes from the client.

Returns:

The only difference between the `MsgRead()` and `MsgRead_r()` functions is the way they indicate errors:

- `MsgRead()`
  - The number of bytes read. If an error occurs, -1 is returned and `errno` is set.

- `MsgRead_r()`
  - The number of bytes read. This function does NOT set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

If you try to read past the end of the thread’s message, the functions return the number of bytes they were actually able to read.

Errors:

- **EFAULT** A fault occurred in a server’s address space when it tried to access the caller’s message buffers.

- **ESRCH** The thread indicated by `rcvid` doesn’t exist or has had its connection detached.

- **ESRVRFault** A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino
**MsgRead()**, **MsgRead_r()**  © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*MsgReadv()*, *MsgReceive()*, *MsgReceivev()*, *MsgReply()*,
*MsgReplyv()*, *MsgWrite()*, *MsgWritev()*
Synopsis:

```c
#include <sys/neutrino.h>

int MsgReadv( int rcvid,  
            const iov_t* riov,  
            int rparts,  
            int offset );

int MsgReadv_r( int rcvid,  
                const iov_t* riov,  
                int rparts,  
                int offset );
```

Arguments:

- **rcvid** The value returned by `MsgReceive*()` when you received the message.
- **riov** An array of buffers where the functions can store the data.
- **rparts** The number of elements in the **riov** array.
- **offset** An offset into the thread’s send message that indicates where you want to start reading the data.

Library:

- **libc**

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `MsgReadv()` and `MsgReadv_r()` kernel calls read data from a message sent by a thread identified by `rcvid`. The thread being read from must not have been replied to and will be in the REPLY-blocked state. Any thread in the receiving process is free to read the message.

These functions are identical except in the way they indicate errors. See the Returns section for details.
The data transfer occurs immediately and the thread doesn’t block. The state of the sending thread doesn’t change.

An attempt to read past the end of the thread’s message results in fewer bytes returned than requested.

You’ll use these functions in these situations:

- A message is sent consisting of a fixed header and a variable amount of data. The header contains the byte count of the data. If the data is large and has to be inserted into one or more buffers (like a filesystem cache), rather than read the data into one large buffer and then copy it into several other buffers, `MsgReceive()` reads only the header, and you can build a custom `iov_t` list to let `MsgReadv()` read data directly into the required buffers.

- A message is received but can’t be handled at the present time. At some point in the future, an event will occur that will allow the message to be processed. Rather than saving the message until it can be processed (thus using memory resources), you can use `MsgReadv()` to reread the message, during which time the sending thread is still blocked.

- Messages that are larger than available buffer space are received. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use `MsgReadv()` to read the message in small pieces, and use `MsgWrite*()` to write the messages in small pieces.

When you’re finished using `MsgReadv()`, you must use `MsgReply*()` to ready the REPLY-blocked process and complete the message exchange.

**Blocking states**

None. In the network case, lower priority threads may run.
Returns:

The only difference between the `MsgReadv()` and `MsgReadv_r()` functions is the way they indicate errors:

- `MsgReadv()` The number of bytes read. If an error occurs, -1 is returned and `errno` is set.
- `MsgReadv_r()` The number of bytes read. This function does **NOT** set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

- **EFAULT** A fault occurred in a server’s address space when it tried to access the caller’s message buffers.
- **ESRCH** The thread indicated by `rcvid` doesn’t exist or has had its connection detached.
- **ESRVRFAULT** A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

September 10, 2007
MsgReadv(), MsgReadv_r() © 2007, QNX Software Systems GmbH & Co. KG.

See also:

MsgRead(), MsgReceive(), MsgReceivev(), MsgReply(), MsgReplyv(),
MsgWrite(), MsgWritev()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgReceive( int chid,
    void * msg,
    int bytes,
    struct _msg_info * info );

int MsgReceive_r( int chid,
    void * msg,
    int bytes,
    struct _msg_info * info );
```

Arguments:

- **chid**: The ID of a channel that you established by calling `ChannelCreate()`.
- **msg**: A pointer to a buffer where the function can store the received data.
- **bytes**: The size of the buffer.
- **info**: NULL, or a pointer to a `_msg_info` structure where the function can store additional information about the message.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgReceive()` and `MsgReceive_r()` kernel calls wait for a message or pulse to arrive on the channel identified by `chid`, and store the received data in the buffer pointed to by `msg`.

These functions are identical, except in the way they indicate errors; see the Returns section for details.
The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The received data isn’t allowed to overflow the receive buffer area provided.

The msg buffer must be big enough to contain a pulse. If it isn’t, the functions indicate an error of EFAULT.

If a message is waiting on the channel when you call `MsgReceive()`, the calling thread doesn’t block, and the message is immediately copied. If a message isn’t waiting, the calling thread enters the RECEIVE-blocked state until a message arrives.

If multiple messages are sent to a channel without a thread waiting to receive them, the messages are queued in priority order.

The thread’s effective priority might change when it receives a message. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

If you pass a non-NULL pointer for `info`, the functions store additional information about the message and the thread that sent it in the `_msg_info` structure that `info` points to. You can get this information later by calling `MsgInfo()`.

On success, `MsgReceive()` and `MsgReceive_r()` return:

>0 A message was received; the returned value is a receive identifier. You’ll use the `rcvid` with other `Msg*()` kernel calls to interact with and reply to the sending thread. `MsgReceive()` changes the state of the sending thread to REPLY-blocked when the message is received. When you use `MsgReply*()` to reply to the received message, the sending thread is made ready again. The `rcvid` encodes the sending thread’s ID and a local connection ID.

0 A pulse was received; `msg` contains a pulse message of type `_pulse`. When a pulse is received, the kernel space allocated
to hold it is immediately released. The \_msg\_info structure isn’t updated.

Don’t reply to a pulse.

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_RECEIVE</td>
<td>There’s no message waiting</td>
</tr>
</tbody>
</table>

Native networking

In networked message-passing transactions, the most noticeable impact is on the server. The server receives the client’s message from the server’s local npm\_-qnet. Note that the receive ID that comes back from \texttt{MsgReceive()} will have some differences, but you don’t need to worry about the format of the receive ID — just treat it as a “magic cookie.”

When the server unblocks from its \texttt{MsgReceive()}, it may or may not have received as much of the message as it would in the local case. This is because of the way that message passing is defined — the client and the server agree on the size of the message transfer area (the transmit parameters passed to \texttt{MsgSend()} on the client end) and the size of the message receive area on the server’s \texttt{MsgReceive()}.

In a local message pass, the kernel would ordinarily limit the size of the transfer to the minimum of both sizes. But in the networked case, the message is received by the client’s npm\_-qnet into its own private buffers and then sent via transport to the remote npm\_-qnet. Since the size of the server’s receive data area can’t be known in advance by the client’s npm\_-qnet when the message is sent, only a fixed maximum size (currently 8K) message is transferred between the client and the server.

This means, for example, that if the client sends 1 Mbyte of data and the server issues a \texttt{MsgReceive()} with a 1-Mbyte data area, then only the number of bytes determined by a network manager would in fact be...
be transferred. The number of bytes transferred to the server is returned via the last parameter to `MsgReceive()` or a call to `MsgInfo()`, specifically the `msglen` member of struct `msg_info`. The client doesn’t notice this, because it’s still blocked.

You can use the following code to ensure that the desired number of bytes are received. Note that this is handled for you automatically when you’re using the resource manager library:

```c
chid = ChannelCreate(_NTO_CHF_SENDER_LEN);
...
rcvid = MsgReceive(chid, msg, nbytes, &info);

/*
   Doing a network transaction and not all
   the message was send, so get the rest...
*/
if (rcvid > 0 && info.srcmsglen > info.msglen &&
    info.msglen < nbytes) {
    int n;

    if((n = MsgRead_r(rcvid, (char *) msg + info.msglen,
                nbytes - info.msglen, info.msglen)) < 0) {
       (MsgError(rcvid, -n);
        continue;
    }
    info.msglen += n;
}
```

**Returns:**

The only difference between `MsgReceive()` and `MsgReceive_r()` is the way they indicate errors. On success, both functions return a positive `rcvid` if they received a message, or 0 if they received a pulse.

If an error occurs:

- `MsgReceive()` returns -1 and sets `errno`.

- `MsgReceive_r()` returns the negative of a value from the Errors section is returned. This function doesn’t set `errno`. 
**Errors:**

- **EFAULT**  
  A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender’s buffers only when `MsgReceive()` is called, a fault could occur in the sender if the sender’s buffers are invalid. If a fault occurs when accessing the sender buffers (only) they’ll receive an EFAULT and `MsgReceive()` won’t unblock.

- **EINTR**  
  The call was interrupted by a signal.

- **ESRCH**  
  The channel indicated by `chid` doesn’t exist.

- **ETIMEDOUT**  
  A kernel timeout unblocked the call. See `TimerTimeout()`.

**Classification:**

- **QNX Neutrino**

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The maximum size for one-part message-pass is $2^{32} - 1$ (SSIZE_MAX).
See also:

ChannelCreate(), _msg_info, MsgInfo(), MsgRead(), MsgReady(),
MsgReceivePulse(), MsgReceivePulsev(), MsgReceivev(),
MsgReply(), MsgReplyv(), MsgSend(), MsgWrite(), MsgWritev(),
pulse, TimerTimeout()
Synopsis:
#include <sys/neutrino.h>

int MsgReceivePulse( int chid,  
void * pulse,   
int bytes,   
struct _msg_info * info );

int MsgReceivePulse_r( int chid,  
void * pulse,   
int bytes,   
struct _msg_info * info );

Arguments:

chid The ID of a channel that you established by calling ChannelCreate().
pulse A pointer to a buffer where the function can store the received data.
bytes The size of the buffer.
info The function doesn’t update this structure, so you typically pass NULL for this argument.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The MsgReceivePulse() and MsgReceivePulse_r() kernel calls wait for a pulse to arrive on the channel identified by chid and place the received data in the buffer pointed to by pulse.

These functions are identical, except in the way they indicate errors; see the Returns section for details.
The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The received data isn’t allowed to overflow the receive buffer area provided.

The pulse buffer must be big enough to contain a pulse. If it isn’t, the functions indicate an error of EFAULT.

If a pulse is waiting on the channel when you call `MsgReceivePulse()`, the calling thread doesn’t block, and the pulse is immediately copied. If a pulse isn’t waiting, the calling thread enters the RECEIVE-blocked state until a pulse arrives.

If multiple pulses are sent to a channel without a thread waiting to receive them, the pulses are queued in priority order.

The thread’s effective priority might change when it receives a pulse. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

On success, `MsgReceivePulse()` and `MsgReceivePulse_r()` return 0 to indicate that they received a pulse. When a pulse is received:

- the kernel space allocated to hold it is immediately released
- pulse contains a pulse message of type _pulse.

Don’t reply to a pulse.

### Blocking states

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_RECEIVE</td>
<td>There’s no pulse waiting.</td>
</tr>
</tbody>
</table>
Returns:

The only difference between \texttt{MsgReceivePulse()} and 
\texttt{MsgReceivePulse \_r()} is the way they indicate errors. On success, 
they both return 0.

If an error occurred:

\begin{itemize}
  \item \texttt{MsgReceivePulse()} returns -1 and sets \texttt{errno}.
  \item \texttt{MsgReceivePulse \_r()} returns the negative of a value from the 
    Errors section. This function \texttt{doesn't} set \texttt{errno}.
\end{itemize}

Errors:

\begin{itemize}
  \item \texttt{EFAULT} A fault occurred when the kernel tried to access the 
    buffers provided. Because the OS accesses the 
    sender’s buffers only when \texttt{MsgReceivePulse()} is 
    called, a fault could occur \textit{in the sender} if the 
    sender’s buffers are invalid. If a fault occurs when 
    accessing the sender buffers (only) they’ll receive 
    an \texttt{EIFault} and \texttt{MsgReceivePulse()} won’t unblock.
  \item \texttt{EINTR} The call was interrupted by a signal.
  \item \texttt{ESRCH} The channel indicated by \texttt{chid} doesn’t exist.
  \item \texttt{ETIMEDOUT} A kernel timeout unblocked the call. See 
    \texttt{TimerTimeout()}. 
\end{itemize}

Classification:

\begin{itemize}
  \item QNX Neutrino
\end{itemize}

\textbf{Safety}

\begin{itemize}
  \item Cancellation point \hspace{1cm} No
  \item Interrupt handler \hspace{1cm} No
\end{itemize}

\textit{continued...}
See also:

MsgDeliverEvent(), MsgReceive(), MsgReceivePulsev(),
MsgReceivev(), MsgSendPulse(), _pulse, TimerTimeout()
Synopsis:

```
#include <sys/neutrino.h>

int MsgReceivePulsev( int chid,  
                       const iov_t * piov,  
                       int parts,  
                       struct _msg_info * info );

int MsgReceivePulsev_r( int chid,  
                       const iov_t * piov,  
                       int parts,  
                       struct _msg_info * info );
```

Arguments:

- **chid**: The ID of a channel that you established by calling `ChannelCreate()`.
- **piov**: An array of buffers where the function can store the received data.
- **parts**: The number of elements in the array.
- **info**: The function doesn’t update this structure, so you typically pass NULL for this argument.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgReceivePulsev()` and `MsgReceivePulsev_r()` kernel calls wait for a pulse to arrive on the channel identified by `chid` and places the received data in the array of buffers pointed to by `piov`.

These functions are identical, except in the way they indicate errors; see the Returns section for details.
The number of bytes transferred is the minimum of that specified by
both the sender and the receiver. The received data isn’t allowed to
overflow the receive buffer area provided.

The first buffer of the IOV (input/output vector) must be big enough to
contain a pulse. If it isn’t, the functions indicate an error of EFAULT.

If a pulse is waiting on the channel when you call
MsgReceivePulsev(), the calling thread doesn’t block, and the pulse is
immediately copied. If a pulse isn’t waiting, the calling thread enters
the RECEIVE-blocked state until a pulse arrives.

If multiple pulses are sent to a channel without a thread waiting to
receive them, the pulses are queued in priority order.

The thread’s effective priority might change when it receives a pulse.
For more information, see “Priority inheritance and messages” in the
QNX Neutrino Microkernel chapter of the System Architecture
guide.

On success, MsgReceivePulsev() and MsgReceivePulsev_r() return 0
to indicate that they received a pulse. When a pulse is received:

- the kernel space allocated to hold it is immediately released
- the IOV’s first buffer contains a pulse message of type _pulse.

**Blocking states**

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_RECEIVE</td>
<td>There’s no pulse waiting.</td>
</tr>
</tbody>
</table>

**Returns:**

The only difference between MsgReceivePulsev() and
MsgReceivePulsev_r() is the way they indicate errors. On success,
they both return 0.

If an error occurs:
MsgReceivePulsev(),
MsgReceivePulsev_r()

- `MsgReceivePulsev()` returns -1 and sets `errno`.
- `MsgReceivePulsev_r()` returns the negative of a value from the Errors section. This function doesn’t set `errno`.

Errors:

EFAULT A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender’s buffers only when `MsgReceivePulsev()` is called, a fault could occur in the sender if the sender’s buffers are invalid. If a fault occurs when accessing the sender buffers (only) they’ll receive an EFAULT and `MsgReceivePulsev()` won’t unblock.

EINTR The call was interrupted by a signal.

ESRCH The channel indicated by `chid` doesn’t exist.

ETIMEDOUT A kernel timeout unblocked the call. See `TimerTimeout()`.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`MsgDeliverEvent()`, `MsgReceive()`, `MsgReceivePulse()`, `MsgReceivev()`, `MsgSendPulse()`, `_pulse.TimerTimeout()`
**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgReceivev( int chid,
                 const iov_t * riov,
                 int rparts,
                 struct _msg_info * info);

int MsgReceivev_r( int chid,
                   const iov_t * riov,
                   int rparts,
                   struct _msg_info * info);
```

**Arguments:**

- **chid**: The ID of a channel that you established by calling `ChannelCreate()`.
- **riov**: An array of buffers where the function can store the received data.
- **rparts**: The number of elements in the array.
- **info**: NULL, or a pointer to a `_msg_info` structure where the function can store additional information about the message.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgReceivev()` and `MsgReceivev_r()` kernel calls wait for a message or pulse to arrive on the channel identified by `chid` and place the received data in the array of buffers pointed to by `riov`.

These functions are identical, except in the way they indicate errors; see the Returns section for details.
The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The received data isn’t allowed to overflow the receive buffer area provided.

The first buffer of the IOV (input/output vector) must be big enough to contain a pulse. If it isn’t, the functions indicate an error of EFAULT.

If a message is waiting on the channel when you call `MsgReceivev()`, the calling thread won’t block, and the message is immediately copied. If a message isn’t waiting, the calling thread enters the RECEIVE-blocked state until a message arrives.

If multiple messages are sent to a channel without a thread waiting to receive them, the messages are queued in priority order.

The thread’s effective priority might change when it receives a message. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

If you pass a non-NULL pointer for `info`, the functions store additional information about the message and the thread that sent it in the `_msg_info` structure that `info` points to. You can get this information later by calling `MsgInfo()`.

On success, `MsgReceivev()` and `MsgReceivev_r()` return:

- >0 A message was received; the value returned is a `rcvid` (receive identifier). You’ll use the `rcvid` with other `Msg*()` kernel calls to interact with and reply to the sending thread. `MsgReceivev()` changes the state of the sending thread to REPLY-blocked when the message is received. When you use `MsgReply*()` to reply to the received message, the sending thread is made ready again. The `rcvid` encodes the sending thread’s ID and a local connection ID.

- 0 A pulse was received; the IOV’s first buffer contains a pulse message of type `_pulse`. When a pulse is received, the kernel
space allocated to hold it is immediately released. The 
_msg_info structure isn’t updated.

Don’t reply to a pulse.

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_RECEIVE</td>
<td>There’s no message waiting.</td>
</tr>
</tbody>
</table>

**Blocking states**

**Returns:**

The only difference between `MsgReceivev()` and `MsgReceivev_r()` is the way they indicate errors. On success, both functions return a positive `rcvid` if they received a message, or 0 if they received a pulse. If an error occurs:

- `MsgReceivev()` returns -1 and sets `errno`.
- `MsgReceivev_r()` returns the negative of a value from the Errors section. This function doesn’t set `errno`.

**Errors:**

- **EFAULT** A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender’s buffers only when `MsgReceivev()` is called, a fault could occur in the sender if the sender’s buffers are invalid. If a fault occurs when accessing the sender buffers (only) they’ll receive an EFAULT and `MsgReceivev()` won’t unblock.

- **EINTR** The call was interrupted by a signal.

- **ESRCH** The channel indicated by `chid` doesn’t exist.

- **ETIMEDOUT** A kernel timeout unblocked the call. See `TimerTimeout()`.
**MsgReceivev()**, **MsgReceivev_r()** © 2007, QNX Software Systems GmbH & Co. KG.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

ChannelCreate(), `_msg_info`, `MsgInfo()`, `MsgRead()`, `MsgReadv()`, `MsgReceive()`, `MsgReceivePulse()`, `MsgReceivePulsev()`, `MsgReply()`, `MsgReplyv()`, `MsgWrite()`, `MsgWritev()`, `_pulse`, `TimerTimeout()`
Synopsis:

```c
#include <sys/neutrino.h>

int MsgReply( int rcvid, 
              int status, 
              const void* msg, 
              int size );

int MsgReply_r( int rcvid, 
                int status, 
                const void* msg, 
                int size );
```

Arguments:

- **rcvid**: The receive ID that `MsgReceive*()` returned when you received the message.
- **status**: The status to use when unblocking the `MsgSend*()` call in the `rcvid` thread.
- **msg**: A pointer to a buffer that contains the message that you want to reply with.
- **size**: The size of the message, in bytes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgReply()` and `MsgReply_r()` kernel calls reply with a message to the thread identified by `rcvid`. The thread being replied to must be in the REPLY-blocked state. Any thread in the receiving process is free to reply to the message, however, it may be replied to only once for each receive.
These functions are identical except in the way they indicate errors. See the Returns section for details.

The \texttt{MsgSend*}() in the \texttt{rcvid} thread unblocks with a return value of \texttt{status}.

The \texttt{MsgSend*} \texttt{r()} functions return negative \texttt{errno} values to indicate failure, so you shouldn’t pass a negative value for the \texttt{status} to \texttt{MsgReply*}(), because the \texttt{MsgSend*} \texttt{r()} functions could interpret it as an error code.

The number of bytes transferred is the minimum of that specified by both the replier and the sender. The reply data isn’t allowed to overflow the reply buffer area provided by the sender.

The data transfer occurs immediately, and the replying task doesn’t block. There’s no need to reply to received messages in any particular order, but you must eventually reply to each message to allow the sending thread(s) to continue execution.

\textbf{Blocking states}

None. In the network case, lower priority threads may run.

\textbf{Native networking}

The \texttt{MsgReply()} function has increased latency when it’s used to communicate across a network — the server is now writing data to its local \texttt{npm-qnet}, which may need to communicate with the client’s \texttt{npm-qnet} to actually transfer the data.

\textbf{Returns:}

The only difference between the \texttt{MsgReply()} and \texttt{MsgReply*r()} functions is the way they indicate errors:

\begin{verbatim}
\texttt{MsgReply()} \quad If an error occurs, -1 is returned and \texttt{errno} is set.
\texttt{MsgReply*r()} \quad This function does NOT set \texttt{errno}. If an error occurs, the negative of a value from the Errors section is returned.
\end{verbatim}
**Errors:**

- **EFAULT**
  A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffers.

- **ESRCH**
  The thread indicated by `rcvid` doesn’t exist, or is no longer REPLY-blocked on the channel, or the connection is detached.

- **ESRVRFault**
  A fault occurred when the kernel tried to access the buffers provided.

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**Caveats:**

The maximum size for one-part message-pass is $2^{32} - 1$ (SSIZE_MAX).

**See also:**

`MsgReceive()`, `MsgReceivev()`, `MsgReply()`, `MsgSend()`, `MsgSendv()`, `MsgWrite()`, `MsgWritev()`
**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgReplyv( int rcvid,
               int status,
               const iov_t* riiov,
               int rparts );

int MsgReplyv_r( int rcvid,
                 int status,
                 const iov_t* riiov,
                 int rparts );
```

**Arguments:**

- **rcvid**: The receive ID that `MsgReceive*()` returned when you received the message.
- **status**: The status to use when unblocking the `MsgSend*()` call in the `rcvid` thread.
- **riiov**: An array of buffers that contains the message that you want to reply with.
- **rparts**: The number of elements in the array.

**Library:**

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `MsgReplyv()` and `MsgReplyv_r()` kernel calls reply with a message to the thread identified by `rcvid`. The thread being replied to must be in the REPLY-blocked state. Any thread in the receiving process is free to reply to the message, however, it may be replied to only once for each receive.
These functions are identical except in the way they indicate errors. See the Returns section for details.

The `MsgSend*()` in the `rcvid` thread unblocks with a return value of `status`.

The `MsgSend*__r()` functions return negative `errno` values to indicate failure, so you shouldn’t pass a negative value for the `status` to `MsgReply*()`, because the `MsgSend*__r()` functions could interpret it as an error code.

The data is taken from the array of message buffers pointed to by `riov`. The number of elements in this array is given by `rparts`. The size of the message is the sum of the sizes of each buffer.

The number of bytes transferred is the minimum of that specified by both the replier and the sender. The reply data isn’t allowed to overflow the reply buffer area provided by the sender.

The data transfer occurs immediately, and the replying task doesn’t block. There’s no need to reply to received messages in any particular order, but you must eventually reply to each message to allow the sending thread(s) to continue execution.

It’s quite common to reply with two-part messages consisting of a fixed header and a buffer of data. The `MsgReplyv()` function gathers the data from the buffer list into a logical contiguous message and transfers it to the sender’s reply buffer(s). The sender doesn’t need to specify the same number or size of buffers. The data is laid down filling each buffer as required. The filesystem, for example, builds a reply list pointing into its cache in order to reply with what appears to be one contiguous piece of data.

**Blocking states**

None. In the network case, lower priority threads may run.
Returns:

The only difference between the `MsgReplyv()` and `MsgReplyv_r()` functions is the way they indicate errors:

- **MsgReplyv()**: If an error occurs, -1 is returned and `errno` is set.
- **MsgReplyv_r()**: This function does **NOT** set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

- **EFAULT**: A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffers.
- **ESRCH**: The thread indicated by `rcvid` doesn’t exist, or is no longer REPLY-blocked on the channel, or the connection is detached.
- **ESRVRFAULT**: A fault occurred when the kernel tried to access the buffers provided.

Classification:

- **QNX Neutrino**

**Safety**

<table>
<thead>
<tr>
<th>Component</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

MsgReceive(), MsgReceivev(), MsgReply(), MsgSend(), MsgSendv(),
MsgWrite(), MsgWritev()
**MsgSend(), MsgSend_r()**

Send a message to a channel

**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgSend( int coid,
             const void* smsg,
             int sbytes,
             void* rmsg,
             int rbytes );

int MsgSend_r( int coid,
               const void* smsg,
               int sbytes,
               void* rmsg,
               int rbytes );
```

**Arguments:**

- **coid**  
The ID of the channel to send the message on, which you've established by calling `ConnectAttach()`.

- **smsg**  
A pointer to a buffer that contains the message that you want to send.

- **sbytes**  
The number of bytes to send.

- **rmsg**  
A pointer to a buffer where the reply can be stored.

- **rbytes**  
The size of the reply buffer, in bytes.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgSend()` and `MsgSend_r()` kernel calls send a message to a process’s channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

.MsgSend() is a cancellation point for the ThreadCancel() kernel call; MsgSendnc() isn’t.

**Blocking states**

<table>
<thead>
<tr>
<th>STATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND</td>
<td>The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.</td>
</tr>
<tr>
<td>REPLY</td>
<td>The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.</td>
</tr>
</tbody>
</table>
Native networking

When a client sends a message to a remote server, the client is effectively sending the message via its local microkernel; the network manager does the actual “work.” The local network manager negotiates with the remote network manager and causes the message to be delivered there. However, the remote manager is the one that actually delivers the message to the server.

This message transfer from the remote manager to the server is accomplished via a special nonblocking message pass.

The only impact on the client is the latency of the message-passing operations. This is purely a function of the network link speed and the overhead associated with the protocol (i.e. npm-qnet for native networking) that io-net uses.

The client still remains blocked in its MsgSend(), and unblocks only on account of a signal, a kernel timeout, or the completion of its function.

Returns:

The only difference between the MsgSend() and MsgSend_r() functions is the way they indicate errors:

\[\text{MsgSend}() \quad \text{Success} \quad \text{The value of status from } \text{MsgReply}().\]
\[-1 \quad \text{An error occurred (errno is set), or the server called } \text{MsgError}() \text{ (errno is set to the error value passed to } \text{MsgError}()).\]

\[\text{MsgSend}_r() \quad \text{Success} \quad \text{The value of status from } \text{MsgReply}().\]
\[\text{negative value} \quad \text{An error occurred (errno is NOT set, the value is the negative of a value from the Errors section), or the server called } \text{MsgError}() \text{ (errno is NOT set, the value is the negative of the error value passed to } \text{MsgError}()).\]
MsgSend(), MsgSend_r()

Errors:

EBADF The connection indicated by coid is no longer connected to a channel, or the connection indicated by coid doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

EINTR The call was interrupted by a signal.

ESRCH The server died while the calling thread was SEND-blocked or REPLY-blocked.

ESRVRFAULT A fault occurred in a server’s address space when accessing the server’s message buffers.

ETIMEDOUT A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
MsgSend(), MsgSend_r()  © 2007, QNX Software Systems GmbH & Co. KG.

Caveats:

The maximum size for one-part message-pass is \(2^{32} - 1\) (SSIZE_MAX).

See also:

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(), MsgReplyv(), MsgSendnc(), MsgSendPulse(), MsgSendsv(), MsgSendsvnc(), MsgSendv(), MsgSendvnc(), MsgSendvs(), MsgSendvsnc(), TimerTimeout()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgSendnc( int coid,  
               const void* smsg,  
               int sbytes,  
               void* rmsg,  
               int rbytes );

int MsgSendnc_r( int coid,  
                const void* smsg,  
                int sbytes,  
                void* rmsg,  
                int rbytes );
```

Arguments:

- **coid**: The ID of the channel to send the message on, which you’ve established by calling `ConnectAttach()`.
- **smsg**: A pointer to a buffer that contains the message that you want to send.
- **sbytes**: The number of bytes to send.
- **rmsg**: A pointer to a buffer where the reply can be stored.
- **rbytes**: The size of the reply buffer, in bytes.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgSendnc()` and `MsgSendnc_r()` kernel calls send a message to a process’s channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

*MsgSend() is a cancellation point for the `ThreadCancel()` kernel call; `MsgSendnc()` isn’t.*

### Blocking states

<table>
<thead>
<tr>
<th>STATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_SEND</td>
<td>The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.</td>
</tr>
<tr>
<td>STATE_REPLY</td>
<td>The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.</td>
</tr>
</tbody>
</table>
**MsgSendnc(),
MsgSendnc_r()**

**Returns:**

The only difference between the `MsgSendnc()` and `MsgSendnc_r()` functions is the way they indicate errors:

- **MsgSendnc()**
  - Success: The value of `status` from `MsgReply*()`.
  - -1: An error occurred (`errno` is set), or the server called `MsgError*()` (`errno` is set to the error value passed to `MsgError()`).

- **MsgSendnc_r()**
  - Success: The value of `status` from `MsgReply*()`.
  - negative value: An error occurred (`errno` is **NOT** set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (`errno` is **NOT** set, the value is the negative of the error value passed to `MsgError()`).

**Errors:**

- **EBADF**
  The connection indicated by `coid` is no longer connected to a channel, or the connection indicated by `coid` doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

- **EFAULT**
  A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

- **EINTR**
  The call was interrupted by a signal.

- **ESRCH**
  The server died while the calling thread was SEND-blocked or REPLY-blocked.

- **ESRVRFault**
  A fault occurred in a server’s address space when accessing the server’s message buffers.
ETIMEDOUT  A kernel timeout unblocked the call. See
TimerTimeout().

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The maximum size for one-part message-pass is $2^{32} - 1$ (SSIZE_MAX).

See also:

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(),
MsgReplyv(), MsgSend(), MsgSendPulse(), MsgSendsv(),
MsgSendsvnc(), MsgSendv(), MsgSendvnc(), MsgSendvs(),
MsgSendvsnc(), TimerTimeout()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgSendPulse ( int coid,
                  int priority,
                  int code,
                  int value );

int MsgSendPulse_r ( int coid,
                     int priority,
                     int code,
                     int value );
```

Arguments:

- **coid**  The ID of the channel to send the message on, which you’ve established by calling `ConnectAttach()`.

- **priority**  The priority to use for the pulse. This must be within the range of valid priorities, which you can determine by calling `sched_get_priority_min()` and `sched_get_priority_max()`.

- **code**  The 8-bit pulse code. Although `code` can be any 8-bit signed value, you should avoid `code` values less than zero, in order to avoid conflict with kernel- or QNX manager-generated pulse codes. These codes all start with `_PULSE_CODE_` and are defined in `<sys/neutrino.h>`; for more information, see the documentation for the `_pulse` structure. A safe range of pulse values is `_PULSE_CODE_MINAVAL` through `_PULSE_CODE_MAXAVAL`.

- **value**  The 32-bit pulse value.
**Library:**

```
libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
```

**Description:**

The `MsgSendPulse()` and `MsgSendPulse_r()` kernel calls send a short, nonblocking message to a process’s channel identified by `coid`.

These functions are identical except in the way they indicate errors. See the Returns section for details.

You can send a pulse to a process if the sending process’s real or effective user ID either:

- matches the real or effective user ID of the receiving process
  - Or:
  - equals zero.

This permission checking is identical to that used by `kill()`.

You can use `MsgSendPulse()` for many purposes; however, due to the small payload of data, you shouldn’t use it for transmitting large amounts of bulk data by sending a great number of pulses.

Pulses are queued for the receiving process in the system, using a dynamic pool of memory objects. If pulses are generated faster than they can be consumed by the receiver, then over a period of time the system queue for the pulses could reach a low memory condition. If there’s no memory available for the pulse to be queued in the system, the kernel fails the pulse request with an error of EAGAIN. If the priority, code and value don’t change, the kernel compresses the pulses by storing an 8-bit count with an already queued pulse.
The receiving thread’s effective priority might change when you send a pulse to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

When you receive a pulse via the `MsgReceive*()` kernel call, the `rcvid` returned is zero. This indicates to the receiver that it’s a pulse and, unlike a message, shouldn’t be replied to using `MsgReply*()`.

In a client/server design, `MsgDeliverEvent()` is typically used in the server, and `MsgSendPulse()` in the client.

### Blocking states

None. In the network case, lower priority threads may run.

### Native networking

You can use `MsgSendPulse()` to send pulses across the network.

### Returns:

The only difference between the `MsgSendPulse()` and `MsgSendPulse_r()` functions is the way they indicate errors:

- **`MsgSendPulse()`**
  - If an error occurs, -1 is returned and `errno` is set. Any other value returned indicates success.

- **`MsgSendPulse_r()`**
  - EOK is returned on success. This function does **NOT** set `errno`. If an error occurs, any value in the Errors section may be returned.
MsgSendPulse(), MsgSendPulse_r() © 2007, QNX Software Systems GmbH & Co. KG.

Errors:

- **EAGAIN**: The kernel had insufficient resources to enqueue the pulse.
- **EBADF**: The connection indicated by `coid` is no longer connected to a channel or the connection indicated by `coid` doesn’t exist. The channel may have been terminated by the server or the network manager if it failed to respond to multiple polls.
- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided.
- **EPERM**: This process doesn’t have sufficient permission to send a pulse to the connection, `coid`.
- **ESRVRFAULT**: A fault occurred in the server’s address space when it tried to write the pulse message to the server’s receive message buffer.

Classification:

- **QNX Neutrino**

Safety

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

If the server faults on delivery, the pulse is either lost or an error is returned.
See also:

*MsgDeliverEvent*, *MsgReceive*, *MsgReceivePulse*,
*MsgSend*, *MsgSendnc*, *MsgSendsv*, *MsgSendsvnc*, *MsgSendv*,
*MsgSendvnc*, *MsgSendvs*, *MsgSendvsnc*, *pulse*,
*sched_get_priority_min*, *sched_get_priority_max*
**MsgSendsv(), MsgSendsv_r()** © 2007, QNX Software Systems GmbH & Co. KG.

Send a message to a channel

**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgSendsv( int coid, 
        const void* smsg, 
        int sbytes, 
        const iov_t* riov, 
        int rparts );

int MsgSendsv_r( int coid, 
        const void* smsg, 
        int sbytes, 
        const iov_t* riov, 
        int rparts );
```

**Arguments:**

- **coid** The ID of the channel to send the message on, which you've established by calling `ConnectAttach()`.
- **smsg** A pointer to a buffer that contains the message that you want to send.
- **sbytes** The number of bytes to send.
- **riov** An array of buffers where the reply can be stored.
- **rparts** The number of elements in the `riov` array.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgSendsv()` and `MsgSendsv_r()` kernel calls send a message to a process’s channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

MsgSendsv() is a cancellation point for the ThreadCancel() kernel call; MsgSendsvnc() isn’t.

**Blocking states**

**STATE_SEND** The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to **STATE_REPLY**.

**STATE_REPLY** The message has been received but not yet replied to. This state may be entered directly, or from **STATE_SEND**.
Returns:

The only difference between the `MsgSendsv()` and `MsgSendsv_r()` functions is the way they indicate errors:

- `MsgSendsv()`: Success, the value of `status` from `MsgReply*()`.
  - `-1`: An error occurred (`errno` is set), or the server called `MsgError*()` (`errno` is set to the error value passed to `MsgError()`).
- `MsgSendsv_r()`: Success, the value of `status` from `MsgReply*()`.
  - `negative value`: An error occurred (`errno` is NOT set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (`errno` is NOT set, the value is the negative of the error value passed to `MsgError()`).

Errors:

- **EBADF**
  - The connection indicated by `coid` is no longer connected to a channel, or the connection indicated by `coid` doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.
- **EFAULT**
  - A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.
- **EINTR**
  - The call was interrupted by a signal.
- **ESRCH**
  - The server died while the calling thread was SEND-blocked or REPLY-blocked.
- **ESRVRFault**
  - A fault occurred in a server’s address space when accessing the server’s message buffers.
**MsgSendsv()**, **MsgSendsv_r()**

**ETIMEDOUT**  A kernel timeout unblocked the call. See `TimerTimeout()`.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`ConnectAttach()`, `MsgReceive()`, `MsgReceivev()`, `MsgReply()`, `MsgReplvy()`, `MsgSend()`, `MsgSendnc()`, `MsgSendPulse()`, `MsgSendsvnc()`, `MsgSendv()`, `MsgSendvnc()`, `MsgSendvs()`, `MsgSendvsnc()`, `TimerTimeout()`
Send a message to a channel (non-cancellation point)

Synopsis:

```
#include <sys/neutrino.h>

int MsgSendsvnc( int coid,  
    const void* smsg,  
    int sbytes,  
    const iov_t* riov,  
    int rparts );

int MsgSendsvnc_r( int coid,  
    const void* smsg,  
    int sbytes,  
    const iov_t* riov,  
    int rparts );
```

Arguments:

- `coid` The ID of the channel to send the message on, which you’ve established by calling `ConnectAttach()`.
- `smsg` A pointer to a buffer that contains the message that you want to send.
- `sbytes` The number of bytes to send.
- `riov` An array of buffers where the reply can be stored.
- `rparts` The number of elements in the `riov` array.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgSendsvnc()` and `MsgSendsvnc_r()` kernel calls send a message to a process’s channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

(MsgSendsv() is a cancellation point for the ThreadCancel() kernel call; MsgSendsvnc() isn’t.

**Blocking states**

**STATE_SEND**

The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.

**STATE_REPLY**

The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.
Returns:

The only difference between the `MsgSendsvnc()` and `MsgSendsvnc_r()` functions is the way they indicate errors:

`MsgSendsvnc()`
- Success: The value of `status` from `MsgReply*()`.
- `-1`: An error occurred (`errno` is set), or the server called `MsgError*()` (`errno` is set to the error value passed to `MsgError()`).

`MsgSendsvnc_r()`
- Success: The value of `status` from `MsgReply*()`.
- Negative value: An error occurred (`errno` is NOT set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (`errno` is NOT set, the value is the negative of the error value passed to `MsgError()`).

Errors:

- **EBADF**: The connection indicated by `coid` is no longer connected to a channel, or the connection indicated by `coid` doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.
- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.
- **EINTR**: The call was interrupted by a signal.
- **ESRCH**: The server died while the calling thread was SEND-blocked or REPLY-blocked.
- **ESRVRFault**: A fault occurred in a server’s address space when accessing the server’s message buffers.
ETIMEDOUT  A kernel timeout unblocked the call. See TimerTimeout().

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(), MsgReplyv(), MsgSend(), MsgSendnc(), MsgSendPulse(), MsgSendsv(), MsgSendv(), MsgSendvnc(), MsgSendvs(), MsgSendvsnc(), TimerTimeout()
**msgsndv(), msgsndv_r()** © 2007, QNX Software Systems GmbH & Co. KG.

Send a message to a channel

**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgSendv( int coid,
               const iov_t* siov,
               int sparts,
               const iov_t* riov,
               int rparts );

int MsgSendv_r( int coid,
                 const iov_t* siov,
                 int sparts,
                 const iov_t* riov,
                 int rparts );
```

**Arguments:**

- **coid** The ID of the channel to send the message on, which you've established by calling `ConnectAttach()`.
- **siov** An array of buffers that contains the message that you want to send.
- **sparts** The number of elements in the `siov` array.
- **riov** An array of buffers where the reply can be stored.
- **rparts** The number of elements in the `riov` array.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgSendv()` and `MsgSendv_r()` kernel calls send a message to a process's channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

It’s quite common to send two-part messages consisting of a fixed header and a buffer of data. The \texttt{MsgSendv()} function gathers the data from the send list into a logically contiguous message and transfers it to the receiver. The receiver doesn’t need to specify the same number or size of buffers. The data is laid down filling each entry as required. The same applies to the replied data.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

\texttt{MsgSendv()} is a cancellation point for the \texttt{ThreadCancel()} kernel call; \texttt{MsgSendvnc()} isn’t.

### Blocking states

| STATE _SEND | The message has been sent but not yet received. If a thread is waiting to receive the message, this |
state is skipped and the calling thread goes directly to STATE_REPLY.

STATE_REPLY The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.

Returns:

The only difference between theMsgSendv() and MsgSendv_r() functions is the way they indicate errors:

\[
\begin{align*}
\text{MsgSendv()} & \quad \text{Success} \quad & \text{The value of status from } \text{MsgReply*}(). \\
& \quad \text{-1} \quad & \text{An error occurred (errno is set), or the server called } \text{MsgError*}() \text{ (errno is set to the error value passed to } \text{MsgError}()). \\
\text{MsgSendv_r()} & \quad \text{Success} \quad & \text{The value of status from } \text{MsgReply*}(). \\
& \quad \text{negative value} \quad & \text{An error occurred (errno is NOT set, the value is the negative of a value from the Errors section), or the server called } \text{MsgError*}() \text{ (errno is NOT set, the value is the negative of the error value passed to } \text{MsgError}()).
\end{align*}
\]

Errors:

EBADF The connection indicated by coid is no longer connected to a channel, or the connection indicated by coid doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

EFAULT A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.
**MsgSendv(), MsgSendv_r()**

EINTR  The call was interrupted by a signal.
ESRCH  The server died while the calling thread was SEND-blocked or REPLY-blocked.
ESRVRFAULT  A fault occurred in a server’s address space when accessing the server’s message buffers.
ETIMEDOUT  A kernel timeout unblocked the call. See TimerTimeout().

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(), MsgReplyv(), MsgSend(), MsgSendn(), MsgSendPulse(), MsgSendsv(), MsgSendsvnc(), MsgSendvnc(), MsgSendvs(), MsgSendvsnc(), TimerTimeout()
Send a message to a channel (non-cancellation point)

Synopsis:

```c
#include <sys/neutrino.h>

int MsgSendvnc(int coid, const iov_t* siov, int sparts, const iov_t* riov, int rparts);

int MsgSendvnc_r(int coid, const iov_t* siov, int sparts, const iov_t* riov, int rparts);
```

Arguments:

- `coid`: The ID of the channel to send the message on, which you've established by calling `ConnectAttach()`.
- `siov`: An array of buffers that contains the message that you want to send.
- `sparts`: The number of elements in the `siov` array.
- `riov`: An array of buffers where the reply can be stored.
- `rparts`: The number of elements in the `riov` array.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgSendvnc()` and `MsgSendvnc_r()` kernel calls send a message to a process’s channel identified by `coid`.
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

MsgSendv() is a cancellation point for the ThreadCancel() kernel call; MsgSendvnc() isn’t.

### Blocking states

<table>
<thead>
<tr>
<th>STATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_SEND</td>
<td>The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.</td>
</tr>
<tr>
<td>STATE_REPLY</td>
<td>The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.</td>
</tr>
</tbody>
</table>
Returns:

The only difference between the `MsgSendvnc()` and `MsgSendvnc_r()` functions is the way they indicate errors:

`MsgSendvnc()`
- **Success**: The value of `status` from `MsgReply*()`.
- **-1**: An error occurred (`errno` is set), or the server called `MsgError*()` (`errno` is set to the error value passed to `MsgError()`).

`MsgSendvnc_r()`
- **Success**: The value of `status` from `MsgReply*()`.
- **negative value**: An error occurred (`errno` is **NOT** set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (`errno` is **NOT** set, the value is the negative of the error value passed to `MsgError()`).

Errors:

- **EBADF**: The connection indicated by `coid` is no longer connected to a channel, or the connection indicated by `coid` doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

- **EINTR**: The call was interrupted by a signal.

- **ESRCH**: The server died while the calling thread was SEND-blocked or REPLY-blocked.

- **ESRVRFAULT**: A fault occurred in a server’s address space when accessing the server’s message buffers.
ETIMEDOUT A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(),
MsgReplyv(), MsgSend(), MsgSendnc(), MsgSendPulse(),
MsgSendsv(), MsgSendsvnc(), MsgSendv(), MsgSendvs(),
MsgSendvsnc(), TimerTimeout()
Send a message to a channel

Synopsis:

```c
#include <sys/neutrino.h>

int MsgSendvs(int coid, const iov_t* siov, int sparts, void* rmsg, int rbytes);

int MsgSendvs_r(int coid, const iov_t* siov, int sparts, void* rmsg, int rbytes);
```

Arguments:

- `coid` The ID of the channel to send the message on, which you’ve established by calling `ConnectAttach()`.
- `siov` An array of buffers that contains the message that you want to send.
- `sparts` The number of elements in the `siov` array.
- `rmsg` A pointer to a buffer where the reply can be stored.
- `rbytes` The size of the reply buffer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgSendvs()` and `MsgSendvs_r()` kernel calls send a message to a process’s channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

MsgSendvs() is a cancellation point for the ThreadCancel() kernel call; MsgSendvsnc() isn’t.

Blocking states

<table>
<thead>
<tr>
<th>STATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_SEND</td>
<td>The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.</td>
</tr>
<tr>
<td>STATE_REPLY</td>
<td>The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.</td>
</tr>
</tbody>
</table>
Returns:

The only difference between the `MsgSendvs()` and `MsgSendvs_r()` functions is the way they indicate errors:

- **`MsgSendvs()`**
  - **Success**: The value of `status` from `MsgReply*()`.
  - **-1**: An error occurred (`errno` is set), or the server called `MsgError*()` (`errno` is set to the error value passed to `MsgError()`).

- **`MsgSendvs_r()`**
  - **Success**: The value of `status` from `MsgReply*()`.
  - **negative value**: An error occurred (`errno` is NOT set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (`errno` is NOT set, the value is the negative of the error value passed to `MsgError()`).

Errors:

- **EBADF**: The connection indicated by `coid` is no longer connected to a channel, or the connection indicated by `coid` doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

- **EINTR**: The call was interrupted by a signal.

- **ESRCH**: The server died while the calling thread was SEND-blocked or REPLY-blocked.

- **ESRVRFAULT**: A fault occurred in a server’s address space when accessing the server’s message buffers.
ETIMEDOUT A kernel timeout unblocked the call. See TimerTimeout().

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(),
MsgReplyv(), MsgSend(), MsgSendnc(), MsgSendPulse(),
MsgSendsv(), MsgSendsvnc(), MsgSendv(), MsgSendvnc(),
MsgSendvsnc(), TimerTimeout()
**MsgSendvsnc(), MsgSendvsnc_r()** © 2007, QNX Software Systems GmbH & Co. KG.

Send a message to a channel (non-cancellation point)

**Synopsis:**
```c
#include <sys/neutrino.h>

int MsgSendvsnc( int coid,
                 const iov_t* siov,
                 int sparts,
                 void* rmsg,
                 int rbytes );

int MsgSendvsnc_r( int coid,
                 const iov_t* siov,
                 int sparts,
                 void* rmsg,
                 int rbytes );
```

**Arguments:**
- `coid` The ID of the channel to send the message on, which you’ve established by calling `ConnectAttach()`.
- `siov` An array of buffers that contains the message that you want to send.
- `sparts` The number of elements in the `siov` array.
- `rmsg` A pointer to a buffer where the reply can be stored.
- `rbytes` The size of the reply buffer.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `MsgSendvsnc()` and `MsgSendvsnc_r()` kernel calls send a message to a process’s channel identified by `coid`. 
These functions are identical except in the way they indicate errors. See the Returns section for details.

The number of bytes transferred is the minimum of that specified by both the sender and the receiver. The send data isn’t allowed to overflow the receive buffer area provided by the receiver. The reply data isn’t allowed to overflow the reply buffer area provided.

The sending thread becomes blocked waiting for a reply. If the receiving process has a thread that’s RECEIVE-blocked on the channel, the transfer of data into its address space occurs immediately, and the receiving thread is unblocked and made ready to run. The sending thread becomes REPLY-blocked. If there are no waiting threads on the channel, the sending thread becomes SEND-blocked and is placed in a queue (perhaps with other threads). In this case, the actual transfer of data doesn’t occur until a receiving thread receives on the channel. At this point, the sending thread becomes REPLY-blocked.

The receiving thread’s effective priority might change when you send a message to it. For more information, see “Priority inheritance and messages” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

MsgSendvs() is a cancellation point for the ThreadCancel() kernel call; MsgSendvsnc() isn’t.

### Blocking states

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_SEND</td>
<td>The message has been sent but not yet received. If a thread is waiting to receive the message, this state is skipped and the calling thread goes directly to STATE_REPLY.</td>
</tr>
<tr>
<td>STATE_REPLY</td>
<td>The message has been received but not yet replied to. This state may be entered directly, or from STATE_SEND.</td>
</tr>
</tbody>
</table>
Returns:

The only difference between the `MsgSendvsnc()` and `MsgSendvsnc_r()` functions is the way they indicate errors:

\[\text{MsgSendvsnc()}\]
- **Success** The value of status from `MsgReply*()`.
- **-1** An error occurred (errno is set), or the server called `MsgError*()` (errno is set to the error value passed to `MsgError()`).

\[\text{MsgSendvsnc_r()}\]
- **Success** The value of status from `MsgReply*()`.
- **negative value** An error occurred (errno is NOT set, the value is the negative of a value from the Errors section), or the server called `MsgError*()` (errno is NOT set, the value is the negative of the error value passed to `MsgError()`).

Errors:

- **EBADF** The connection indicated by coid is no longer connected to a channel, or the connection indicated by coid doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.

- **EFAULT** A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.

- **EINTR** The call was interrupted by a signal.

- **ESRCH** The server died while the calling thread was SEND-blocked or REPLY-blocked.

- **ESRVRFault** A fault occurred in a server’s address space when accessing the server’s message buffers.
MsgSendvsnc(),
MsgSendvsnc_r()

ETIMEDOUT  A kernel timeout unblocked the call. See TimerTimeout().

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(),
MsgReplyv(), MsgSend(), MsgSendnc(), MsgSendPulse(),
MsgSendsv(), MsgSendsvnc(), MsgSendv(), MsgSendvnc(),
MsgSendvs(), TimerTimeout()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgVerifyEvent( int rcvid,
                    const struct sigevent event );

int MsgVerifyEvent_r( int rcvid,
                       const struct sigevent event );
```

Arguments:

- `rcvid` - The receive ID that you want to check.
- `event` - A pointer to a `sigevent` structure that contains the event you want to check.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgVerifyEvent()` and `MsgVerifyEvent_r()` kernel calls check the validity of the receive ID `rcvid`, and the `event` configuration. You can use these functions to verify that an event is well-formed by a client (pass a `rcvid` of 0), and by a server (pass a `rcvid` of the target thread).

These functions are identical except in the way they indicate errors. See the Returns section for details.

Blocking states

These calls don’t block.
Returns:

The only difference between the `MsgVerifyEvent()` and `MsgVerifyEvent_r()` functions is the way they indicate errors:

`MsgVerifyEvent()`

If an error occurs, -1 is returned and `errno` is set.

`MsgVerifyEvent_r()`

This function does **NOT** set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

- **EBADF** The channel for the pulse delivery doesn’t exist.
- **EINVAL** Invalid event structure.
- **ESRCH** The connection for the pulse doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`MsgReceive()`, `MsgReceivev()`, `MsgReply()`, `MsgSend()`, `MsgSendv()`,
`MsgWrite()`, `MsgWritev()`, `sigevent`
MsgWrite(), MsgWrite_r()

Write a reply

Synopsis:

```c
#include <sys/neutrino.h>

int MsgWrite( int rcvid,  
              const void* msg,  
              int size,  
              int offset );

int MsgWrite_r( int rcvid,  
                const void* msg,  
                int size,  
                int offset );
```

Arguments:

- **rcvid** The value returned by `MsgReceive*()` when you received the message.
- **msg** A pointer to a buffer that contains the data you want to write.
- **size** The number of bytes that you want to write. These functions don’t let you write past the end of the sender’s buffer; they return the number of bytes actually written.
- **offset** An offset into the sender’s buffer that indicates where you want to start writing the data.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgWrite()` and `MsgWrite_r()` kernel calls write data to the reply buffer of a thread identified by `rcvid`. The thread being written to must be in the REPLY-blocked state. Any thread in the receiving process is free to write to the reply message.
These functions are identical except in the way they indicate errors. See the Returns section for details.

You use this function in one of these situations:

- The data arrives over time and is quite large. Rather than buffer all the data, you can use `MsgWrite()` to write it into the destination thread’s reply message buffer, as it arrives.

- Messages are received that are larger than available buffer space. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use `MsgRead*()` to read messages in small pieces, and use `MsgWrite()` to write messages in small pieces.

To complete a message exchange, you must call `MsgReply*()`. The reply doesn’t need to contain any data. If it does contain data, then the data is always written at offset zero in the destination thread’s reply message buffer. This is a convenient way of writing the header once all of the information has been gathered.

A single call to `MsgReply*()` is always more efficient than calls to `MsgWrite()` followed by a call to `MsgReply*()`.

**Blocking states**

None. In the network case, lower priority threads may run.

**Native networking**

The `MsgWrite()` function has increased latency when you use it to communicate across a network — the server is now writing data to its local `npm-qnet`, which may need to communicate with the client’s `npm-qnet` to actually transfer the data. The server’s `MsgWrite()` call effectively sends a message to the server’s `npm-qnet` to initiate this data transfer.

But since the server’s `npm-qnet` has no way to determine the size of the client’s receive data area, the number of bytes reported as having been transferred by the server during its `MsgWrite()` call might not be accurate — the reported number will instead reflect the number of bytes transferred by the server to its `npm-qnet`. 
The message is buffered in the server side’s npm-qnet until the client replies, in order to reduce the number of network transactions.

If you want to determine the size of the sender’s reply buffer, set the _NTO_CHF_REPLY_LEN when you call ChannelCreate().

**Returns:**

The only difference between the `MsgWrite()` and `MsgWrite_r()` functions is the way they indicate errors:

- `MsgWrite()` The number of bytes written. If an error occurs, -1 is returned and `errno` is set.
- `MsgWrite_r()` The number of bytes written. This function does **NOT** set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

**Errors:**

- **EFAULT** A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffer.
- **ESRCH** The thread indicated by `rcvid` doesn’t exist or its connection was detached.
- **ESRVRFault** A fault occurred when the kernel tried to access the buffers provided.

**Classification:**

QNX Neutrino

**Safety**

Cancellation point  No

*continued...*
MsgWrite(), MsgWrite_r()

Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ChannelCreate(), MsgRead(), MsgReadv(), MsgReceive(),
MsgReceivev(), MsgReply(), MsgReplyv(), MsgWritev()
Synopsis:

```c
#include <sys/neutrino.h>

int MsgWritev( int rcvid,
        const iov_t* iov,
        int parts,
        int offset );

int MsgWritev_r( int rcvid,
        const iov_t* iov,
        int parts,
        int offset );
```

Arguments:

- `rcvid` The value returned by `MsgReceive*()` when you received the message.
- `iov` An array of buffers that contains the data you want to write.
- `parts` The number of elements in the array. These functions don’t let you write past the end of the sender’s buffer; they return the number of bytes actually written.
- `offset` An offset into the sender’s buffer that indicates where you want to start writing the data.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `MsgWritev()` and `MsgWritev_r()` kernel calls write data to the reply buffer of a thread identified by `rcvid`. The thread being written to must be in the REPLY-blocked state. Any thread in the receiving process is free to write to the reply message.
These functions are identical except in the way they indicate errors. See the Returns section for details.

The data transfer occurs immediately and your thread doesn’t block. The state of the sending thread doesn’t change.

You’ll use this function in one of these situations:

- The data arrives over time and is quite large. Rather than buffer all the data, you can use \textit{MsgWritev()} to write it into the destination thread’s reply message buffer, as it arrives.

- Messages are received that are larger than available buffer space. Perhaps the process is an agent between two processes and simply filters the data and passes it on. You can use \textit{MsgRead*()} to read messages in small pieces, and use \textit{MsgWritev()} to write messages in small pieces.

To complete a message exchange, you must call \textit{MsgReply*()}. The reply doesn’t need to contain any data. If it does contain data, then the data is always written at offset zero in the destination thread’s reply message buffer. This is a convenient way of writing the header once all of the information has been gathered.

A single call to \textit{MsgReply*()} is always more efficient than calls to \textit{MsgWritev()} followed by a call to \textit{MsgReply*()}.

**Blocking states**

None. In the network case, lower priority threads may run.

**Returns:**

The only difference between the \textit{MsgWritev()} and \textit{MsgWritev\_r()} functions is the way they indicate errors:

\textbf{\textit{MsgWritev()}} \hspace{1cm} The number of bytes written. If an error occurs, -1 is returned and \textit{errno} is set.

\textbf{\textit{MsgWritev\_r()}} \hspace{1cm} The number of bytes written. This function does NOT set \textit{errno}. If an error occurs, the negative of a value from the Errors section is returned.
Errors:

- **EFAULT**: A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffer.
- **ESRCH**: The thread indicated by `rcvid` doesn’t exist or its connection was detached.
- **ESRVRFAULT**: A fault occurred when the kernel tried to access the buffers provided.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`MsgRead()`, `MsgReadv()`, `MsgReceive()`, `MsgReceivev()`, `MsgReply()`, `MsgReplyv()`, `MsgWrite()`
Synopsis:

```c
#include <sys/mman.h>

int msync( void * addr,
            size_t len,
            int flags );
```

Arguments:

- `addr` The beginning of the range of addresses that you want to synchronize.
- `len` The length of the range of addresses, in bytes.
- `flags` A bitwise inclusive OR of one or more of the following flags:
  - `MS_ASYNC` — perform asynchronous writes. The function returns immediately once all the write operations are initiated or queued for servicing.
  - `MS_CACHE_ONLY` — a QNX Neutrino extension. See the Caveats section.
    - This flag was added in the QNX Neutrino Core OS 6.3.2.
    - `MS_INVALIDATE` — invalidate cached data. Invalidates all cached copies of mapped data that are inconsistent with the permanent storage locations such that subsequent references obtain data that was consistent with the permanent storage locations sometime between the call to `msync()` and the first subsequent memory reference to the data.
    - `MS_INVALIDATE_ICACHE` — (QNX Neutrino extension) if you’re dynamically modifying code, use this flag to make sure that the new code is what will be executed.
    - `MS_SYNC` — perform synchronous writes. The function doesn’t return until all write operations are completed as defined for synchronized I/O data integrity completion.
msync()

You can specify at most one of MS_ASYNC and MS_SYNC, not both.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The msync() function writes all modified data to permanent storage locations, if any, in those whole pages containing any part of the address space of the process starting at address addr and continuing for len bytes. The msync() function is used with memory mapped files. If no such storage exists, msync() need not have any effect. If requested, the msync() function then invalidates cached copies of data.

For mappings to files, this function ensures that all write operations are completed as defined for synchronized I/O data integrity completion.

Mappings to files aren’t implemented on all filesystems.

When the msync() function is called on MAP_PRIVATE mappings, any modified data won’t be written to the underlying object and won’t cause such data to be made visible to other processes.

The behavior of msync() is unspecified if the mapping wasn’t established by a call to mmap().

If msync() causes any write to a file, the file’s st_ctime and st_mtime fields are marked for update.

For POSIX support of memory-mapped files, the msync() function performs its intended operation of flushing (MS_SYNC or MS_ASYNC), invalidating (MS_INVALIDATE) the data cache or invalidating (MS_INVALIDATE_ICACHE) the instruction cache. This is regardless of the underlying object that is used for the
msync()

mapping (for example, shared memory object, memory-mapped file, anonymous memory, etc). This POSIX behavior of syncing/invalidating to the underlying permanent locations only takes place with memory-mapped file.

Returns:

0  Success
-1  An error occurred (errno is set).

Errors:

EBUSY  Some or all of the addresses in the range starting at addr and continuing for len bytes are locked, and MS_INVALIDATE is specified.
EINVAL  Invalid flags value.
ENOMEM  The addresses in the range starting at addr and continuing for len bytes are outside the range allowed for the address space of a process or specify one or more pages that aren’t mapped.

Classification:

POSIX 1003.1 MF SIO

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
Caveats:

MS_INVALIDATE_ICACHE is a QNX Neutrino extension.

MS_CACHE_ONLY flag ensures that the QNX extension behavior takes place, but not the POSIX standard behavior. This flag performs data cache operation in conjunction with MS_SYNC, MS_ASYNC, or MS_INVALIDATE.

See also:

mmap(), sysconf()
munlock()

Unlock the range of process address space already allocated

Synopsis:

```c
#include <sys/mman.h>

int munlock(const void * addr,  
             size_t len);
```

Arguments:

- `addr`: The starting address for the range of process address space.
- `len`: The length of the memory buffer, in bytes.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `munlock()` function unlocks the pages already locked by `mlock()` or `mlockall()`. It ignores how many times `mlock()` has been called before.

---

The full POSIX implementation for this function was added in the QNX Neutrino Core OS 6.3.2.

---

The `munlock()` function doesn’t affect those pages that have been mapped and locked into the address spaces of other processes.

---

`Memory-resident` is a term used to indicate that the addresses always reside in physical memory.
Returns:

0    Success.
-1   An error has occurred (errno is set).

Errors:

ENOMEM    Some or all of the address range specified by the addr and len arguments doesn’t correspond to valid mapped pages in the address space of the process.

Classification:

POSIX 1003.1 MLR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

mlock(), mlockall()
munlockall()
Unlock a process's address space

Synopsis:

```c
#include <sys/mman.h>

int munlockall( void );
```

Library:

```
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.
```

Description:

The current implementation of the `munlockall()` function doesn’t do anything.

Returns:

-1 to indicate an error (`errno` is set).

Errors:

```
ENOSYS   The `munlockall()` function isn’t currently supported.
```

Classification:

```
POSIX 1003.1 ML
```

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

\textit{mlock()}, \textit{munlock()}, \textit{mlockall()}
munmap()

Unmap previously mapped addresses

Synopsis:

```
#include <sys/mman.h>

int munmap( void * addr,
            size_t len );
```

Arguments:

- `addr` The beginning of the range of addresses that you want to unmap.
- `len` The length of the range of addresses, in bytes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `munmap()` function removes any mappings for pages in the address range starting at `addr` and continuing for `len` bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process. If there are no mappings in the specified address range, then `munmap()` has no effect.

Returns:

- `0` Success.
- `-1` Failure; `errno` is set.

Errors:

- `EINVAL` The addresses in the specified range are outside the range allowed for the address space of a process.
munmap()

The memory manager fails to allocate memory to handle a user’s munmap() request. This allocation of memory is necessary for internal structures to represent the new state of mapped memory.

ENOSYS The function munmap() isn’t supported by this implementation.

Classification:

POSIX 1003.1 MF | SHM | TYM

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Currently, you can’t munmap() just a part of an area mapped with mmap().

See also:

mmap(), mprotect(), shm_open(), shm_unlink()
munmap_device_io()
Free access to a device’s registers

Synopsis:
#include <sys/mman.h>

int munmap_device_io( uintptr_t io,
                      size_t len );

Arguments:
io The address of the area that you want to unmap.
len The number of bytes of device I/O memory that you want to unmap.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The function munmap_device_io() unmaps len bytes of device I/O memory at io (that was previously mapped with mmap_device_io()).

Returns:
-1 An error occurred (errno is set).
Any other value Successful unmapping.

Errors:
EINVAL The addresses in the specified range are outside the range allowed for the address space of a process.
ENOSYS The function munmap() isn’t supported by this implementation.
ENXIO The address from io for len bytes is invalid.
munmap_device_io()

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
munmap()
munmap_device_memory()
Unmap previously mapped addresses

Synopsis:
#include <sys/mman.h>

int munmap_device_memory( void * addr,
                          size_t len );

Arguments:
addr The beginning of the range of addresses that you want to unmap.

len The length of the range of addresses, in bytes.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The munmap_device_memory() function is essentially the same as munmap(). It removes any mappings for pages in the address range starting at addr and continuing for len bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process.

If there are no mappings in the specified address range, then munmap() has no effect.

This function is the complement of mmap_device_memory().

Returns:
-1 An error occurred (errno is set).

Any other value
Success.
**munmap_device_memory()** © 2007, QNX Software Systems GmbH & Co. KG.

**Errors:**

- **EINVAL** The addresses in the specified range are outside the range allowed for the address space of a process.
- **ENOSYS** The `munmap()` function isn’t supported by this implementation.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`munmap_device_memory()`, `munmap()`, `munmap_device_io()`
Unmap previously mapped addresses exercising more control

**Synopsis:**
#include <sys/mman.h>

int munmap( void * addr, size_t len, unsigned flags);

**Arguments:**

- `addr` The beginning of the range of addresses that you want to unmap.
- `len` The length of the range of addresses, in bytes.
- `flags` Use the following values:
  - 0 Behave the same as `munmap()`.
  - `UNMAP_INT_REQUIRED` POSIX Initialization of the page to all zeros is required the next time the underlying physical memory is allocated.
  - `UNMAP_INT_OPTIONAL` Initialization of the underlying physical memory to zeros on its next allocation is optional.

**Library:**
libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `munmap_flags()` removes any mappings for pages in the address range starting at `addr` and continuing for `len` bytes, rounded up to the next multiple of the page size. Subsequent references to these pages cause a SIGSEGV signal to be set on the process.
If there are no mappings in the specified address range, then `munmap_flags()` has no effect.

This function was added in the QNX Neutrino Core OS 6.3.2.

There are some interactions of `flags` argument with MAP_NOINIT flag of the `mmap()` function as well as `procnto` command line as detailed below:

- The `mmap()` function has a new flag, MAP_NOINIT. When specified, the POSIX requirement that the memory be zeroed is relaxed. The physical memory being used for this allocation must have been previously freed with UNMAP_INIT_OPTIONAL for this flag to have any effect.

- The `procnto` command line now has an -mmemmgr_configuration option. The memmgr_configuration string has a sequence of configuration options to enable (or if preceded with a ~ character, disable) memory-manager aspects. The configuration options are:

  1. `munmap_flags()` (with `flags` parameter as zero) acts as if UNMAP_INIT_REQUIRED were specified (the default).
  2. `munmap_flags()` (with `flags` parameter as zero) acts as if UNMAP_INIT_OPTIONAL were specified.

Returns:

0 Success.

-1 Failure; `errno` is set.

Errors:

EINVAL The addresses in the specified range are outside the range allowed for the address space of a process.

ENOSYS The function `munmap_flags()` isn’t supported by this implementation.
munmap_flags()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`mmap()`, `mprotect()`, `munmap()`, `shm_open()`, `shm_unlink()`
Register a name in the namespace and create a channel

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

name_attach_t * name_attach( dispatch_t * dpp, 
const char * path, 
unsigned flags );
```

Arguments:

- `dpp` NULL, or a dispatch handle returned by `dispatch_create()`.
- `path` The path that you want to register under 
`/dev/name/{local|global}/`. This name shouldn’t contain any .. characters or start with a leading slash /.
- `flags` Flags that affect the function’s behavior:
  - `NAME_FLAG_ATTACH_GLOBAL` — attach the name globally instead of locally. See the `gns` utility.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `name_attach()`, `name_close()`, `name_detach()`, and `name_open()` functions provide the basic pathname-to-server-connection mapping, without having to become a full resource manager.

If you’ve already created a dispatch structure, pass it in as the `dpp`. If you provide your own `dpp`, set `flags` to `NAME_FLAG_DETACH_SAVEDPP` when calling `name_detach()`; otherwise, your `dpp` is detached and destroyed automatically.

If you choose to pass a NULL as the `dpp`, `name_attach()` calls `dispatch_create()` and `resmgr_attach()` internally to create a channel, however, it doesn’t set any channel flag by itself.
The `name_attach()` function puts the name `path` into the path namespace under `/dev/name/[local|global]/path`. The name is attached locally by default, or globally when you set `NAME_FLAG_ATTACH_GLOBAL` in the `flags`. You can see attached names in `/dev/name/local` and `/dev/name/global` directories.

The application that calls `name_attach()` receives an `IO_CONNECT` message when `name_open()` is called. The application has to handle this message properly with a reply of an EOK to allow `name_open()` connect.

If the receive buffer that the server provides isn’t large enough to hold a pulse, then `MsgReceive()` returns -1 with `errno` set to EFAULT.

`name_attach_t` The `name_attach()` function returns a pointer to a `name_attach_t` structure that looks like this:

```c
typedef struct _name_attach {
    dispatch_t* dpp;
    int chid;
    int mntid;
    int zero[2];
} name_attach_t;
```

The members include:

- `dpp` The dispatch handle used in the creation of this connection.
- `chid` The channel ID used for `MsgReceive()` directly.
- `mntid` The mount ID for this name.

The information that’s generally required by a server using these services is the `chid`. 
The `gns` utility is a new utility for Global Name Service Manager. It must be started and running first before an application can call `name_attach()` to advertise (or attach) a service (i.e. represented by a path name) either locally or globally.

If an application attaches a service locally, then applications from another machine can’t lookup this service through the `gns` utility. If an application attaches its services globally, then any machine that’s on the network and is running the `gns` manager can access the services.

An application can attach a service locally, only if there isn’t another application that’s attached locally to the same service. There’s no credential restriction for applications that are attached as local services. An application can attach a service globally only if the application has `root` privilege.

**Returns:**

A pointer to a filled-in `name_attach_t` structure, or NULL if the call fails (`errno` is set).

**Errors:**

- **EINVAL** Invalid arguments (i.e. a NULL or empty path, a path starts with a leading slash `/` or contains `..` characters).
- **ENOMEM** Not enough free memory to complete the operation.
- **ENOTDIR** A component of the pathname wasn’t a directory entry.

**Examples:**

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/dispatch.h>

#define ATTACH_POINT "myname"

/* We specify the header as being at least a pulse */
```
typedef struct _pulse msg_header_t;

/* Our real data comes after the header */
typedef struct _my_data {
    msg_header_t hdr;
    int data;
} my_data_t;

/*** Server Side of the code ***/
int server() {
    name_attach_t *attach;
    my_data_t msg;
    int rcvid;

    /* Create a local name (/dev/name/local/...) */
    if ((attach = name_attach(NULL, ATTACH_POINT, 0)) == NULL) {
        return EXIT_FAILURE;
    }

    /* Do your MsgReceive’s here now with the chid */
    while (1) {
        rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);

        if (rcvid == -1) {/* Error condition, exit */
            break;
        }

        if (rcvid == 0) {/* Pulse received */
            switch (msg.hdr.code) {
                case _PULSE_CODE_DISCONNECT:
                    /*
                    * A client disconnected all its connections (called
                    * name_close() for each name_open() of our name) or
                    * terminated
                    */
                    ConnectDetach(msg.hdr.scoid);
                    break;
                case _PULSE_CODE_UNBLOCK:
                    /*
                    * REPLY blocked client wants to unblock (was hit by
                    * a signal or timed out). It’s up to you if you
                    * reply now or later.
                    */
                    break;
                default:
                    /*
                    * A pulse sent by one of your processes or a
                    * _PULSE_CODE_COIDDEATH or _PULSE_CODE_THREADDEATH
                    * from the kernel
                    */
                    break;
            }
        }
    }
}
name_open() sends a connect message, must EOK this */
if (msg.hdr.type == _IO_CONNECT) {
    MsgReply(rcvid, EOK, NULL, 0);
    continue;
}

/* Some other QNX IO message was received; reject it */
if (msg.hdr.type > _IO_BASE && msg.hdr.type <= _IO_MAX) {
    MsgError(rcvid, ENOSYS);
    continue;
}

/* A message (presumably ours) received, handle */
printf("Server receive %d \n", msg.data);
MsgReply(rcvid, EOK, 0, 0);

/* Remove the name from the space */
name_detach(attach, 0);
return EXIT_SUCCESS;

/*** Client Side of the code ***/
int client() {
    my_data_t msg;
    int fd;

    if ((fd = name_open(ATTACH_POINT, 0)) == -1) {
        return EXIT_FAILURE;
    }

    /* We would have pre-defined data to stuff here */
    msg.hdr.type = 0x00;
    msg.hdr.subtype = 0x00;

    /* Do whatever work you wanted with server connection */
    for (msg.data = 0; msg.data < 5; msg.data++) {
        printf("Client sending %d \n", msg.data);
        if (MsgSend(fd, &msg, sizeof(msg), NULL, 0) == -1) {
            break;
        }
    }

name_attach()

/* Close the connection */
name_close(fd);
return EXIT_SUCCESS;
}

int main(int argc, char **argv) {
    int ret;

    if (argc < 2) {
        printf("Usage %s -s | -c \n", argv[0]);
        ret = EXIT_FAILURE;
    } else if (strcmp(argv[1], "-c") == 0) {
        printf("Running Client ... \n");
        ret = client(); /* see name_open() for this code */
    } else if (strcmp(argv[1], "-s") == 0) {
        printf("Running Server ... \n");
        ret = server(); /* see name_attach() for this code */
    } else {
        printf("Usage %s -s | -c \n", argv[0]);
        ret = EXIT_FAILURE;
    }
    return ret;
}

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
name_attach()

© 2007, QNX Software Systems GmbH & Co. KG.

Caveats:

As a server, you shouldn’t assume that you’re doing a MsgReceive() on a clean channel. In QNX Neutrino (and QNX 4), anyone can create a random message and send it to a process or a channel.

We recommend that you do the following to assure that you’re playing safely with others in the system:

```c
#include <sys/neutrino.h>

/* All of your messages should start with this header */
typedef struct _pulse msg_header_t;

/* Now your real data comes after this */
typedef struct _my_data {
    msg_header_t hdr;
    int data;
} my_data_t;
```

where:

- `hdr` Contains a `type/subtype` field as the first 4 bytes. This allows you to identify data which isn’t destined for your server.

- `data` Specifies the receive data structure. The structure must be large enough to contain at least a pulse (which conveniently starts with the `type/subtype` field of most normal messages), because you’ll receive a disconnect pulse when clients are detached.

See also:

`dispatch*()` functions, `MsgReceive()`, `name_detach()`, `name_open()`, `name_close()`, `_pulse`, `resmgr_attach()`
**name_close()**

Close the file descriptor returned by name_open()

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int name_close( int filedes );
```

**Arguments:**

- **filedes**  
The file descriptor returned by name_open().

**Library:**

- **libc**

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `name_close()` function closes the `filedes` obtained with the `name_open()` call.

**Returns:**

Zero for success, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EBADF**  
  Invalid file descriptor `filedes`.
- **EINTR**  
  The `name_close()` call was interrupted by a signal.
- **ENOSYS**  
  The `name_close()` function isn’t implemented for the filesystem specified by `filedes`.

**Examples:**

See the “Client side of the code” section in `name_attach()`.
name_close()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

close(), ConnectDetach(), name_attach(), name_detach(), name_open()
Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int name_detach( name_attach_t * attach,
                 unsigned flags );
```

Arguments:

- `attach` A pointer to the `name_attach_t` structure returned by `name_attach()`.  
- `flags` Flags that affect the function’s behavior:  
  - `NAME_FLAG_DETACH_SAVEDPP` — don’t destroy the dispatch handle.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `name_detach()` function removes the name from the namespace and destroys the channel created by `name_attach()`. If you set `NAME_FLAG_DETACH_SAVEDPP` in `flags`, the dispatch pointer contained in the `name_attach_t` structure isn’t destroyed; it’s up to you to destroy it by calling `dispatch_destroy()`. The default is to destroy the dispatch pointer.

Returns:

Zero on success, or -1 if an error occurs (`errno` is set).
name_detach()

Errors:

EINVAL The mount ID (mntid) was never attached with name_attach().

Examples:

See name_attach() and resmgr_detach().

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ChannelDestroy(), dispatch*() functions, name_attach(), name_close(), name_open(), resmgr_detach()
**name_open()**

Open a name for a server connection

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int name_open( const char * name,
               int flags );
```

**Arguments:**

- `name`  The name that you want to open for a server connection.
- `flags` Flags that affect the function’s behavior:
  - `NAME_FLAG_ATTACH_GLOBAL` — attach the name globally instead of locally.

**Library:**

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `name_open()` function opens `name` for a server connection. No ordering is guaranteed when accessing resources on other nodes.

Before, when an application used to call `name_open()` to connect to a service, the server was not aware of that. This has been changed now — a `_IO_CONNECT/_IO_CONNECT_OPEN` message is actually sent to the server.

The server application has to be modified to handle a possible `_IO_CONNECT` message coming in. See the example code in `name_attach()` to see how to handle this message.
name_open()

Returns:

A nonnegative integer representing a side-channel connection ID (see ConnectAttach()) or -1 if an error occurred (errno is set).

Errors:

EACCES Search permission is denied on a component of the name.

EBADFSYS While attempting to open the named file, either the file itself or a component of the path prefix was found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You’ll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY The connection specified by name has already been opened and additional connections aren’t permitted.

EINTR The name_open() operation was interrupted by a signal.

EISDIR The named path is a directory.

ELOOP Too many levels of symbolic links or prefixes.

EMFILE Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the name string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENFILE Too many files are currently open in the system.

ENOENT The connection specified by name doesn’t exist.

ENOTDIR A component of the name prefix isn’t a directory.
Examples:

See `name_attach()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`ConnectAttach()`, `name_attach()`, `name_detach()`, `name_close()`, `open()`
**Synopsis:**

```c
#include <time.h>

int nanosleep( const struct timespec* rqtp,
               struct timespec* rmtip );
```

**Arguments:**

- `rqtp` A pointer to a `timespec` structure that specifies the time interval for which you want to suspend the thread.
- `rmtip` NULL, or a pointer to a `timespec` structure where the function can store the amount of time remaining in the interval (the requested time minus the time actually slept).

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `nanosleep()` function causes the calling thread to be suspended from execution until either:

- The time interval specified by the `rqtp` argument has elapsed
  - Or
- A signal is delivered to the thread, and the signal’s action is to invoke a signal-catching function or terminate the process.

The suspension time may be longer than requested because the argument value is rounded up to be a multiple of the system timer resolution or because of scheduling and other system activity.
Returns:

- 0  The requested time has elapsed.
- -1 The `nanosleep()` function was interrupted by a signal (`errno` is set).

Errors:

- EAGAIN  All timers are in use. You’ll have to wait for a process to release one.
-EFAULT  A fault occurred trying to access the buffers provided.
- EINTR  The `nanosleep()` function was interrupted by a signal.
- EINVAL  The number of nanoseconds specified by the `tv_nsec` member of the `timespec` structure pointed to by `rqtp` is less than zero or greater than or equal to 1000 million.

Classification:

POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno`, `clock_getres()`, `clock_gettime()`, `clock_settime()`, `sleep()`, `timer_create()`, `timer_delete()`, `timer_gettime()`, `timer_settime()`, `timespec`
nanospin()  
Busy-wait without thread blocking for a period of time

Synopsis:

```c
#include <time.h>

int nanospin( const struct timespec *when );
```

Arguments:

`when`  
A pointer to a `timespec` structure that specifies the amount of time to busy-wait for. This is a duration, not an absolute time.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `nanospin()` function occupies the CPU for the amount of time specified by the argument `when` without blocking the calling thread. (The thread isn’t taken off the ready list.) The function is essentially a `do...while` loop.

The `nanospin*()` functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX `timer_*()` functions.

The first time you call `nanospin()`, the C library invokes `nanospin_calibrate()` with an argument of 0 (interrupts enabled), if you haven’t already called it.

Returns:

- **EOK** Success.
- **E2BIG** The delay specified by `when` is greater than 500 milliseconds.
A delay of more than a few milliseconds might not work on some processors. For longer delays, use the POSIX `timer_*()` functions.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINTR</td>
<td>A too-high rate of interrupts occurred during the calibration routine.</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>This system’s <code>startup-*</code> program didn’t initialize the timing information necessary to use <code>nanospin()</code></td>
</tr>
</tbody>
</table>

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

You should use busy-waiting only when absolutely necessary for accessing hardware.

It isn’t safe to call this function in an interrupt handler if `nanospin_calibrate()` hasn’t been called yet.

**See also:**

`nanosleep()`, `nanospin_calibrate()`, `nanospin_count()`, `nanospin_ns()`, `nanospin_ns_to_count()`, `sched_yield()`, `sleep()`, `timer_create()`, `timespec`
**Synopsis:**

```c
#include <time.h>

int nanospin_calibrate( int disable );
```

**Arguments:**

- `disable`  
  1 to disable interrupts during the call to `nanospin_calibrate()`, or 0 to enable them; see below.

**Library:**

- `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `nanospin_calibrate()` function performs the calibration for the `nanospin()`* family of delay functions. The first time that you call `nanospin()`, `nanospin_ns()`, or `nanospin_ns_to_count()`, the C library invokes `nanospin_calibrate()` with an argument of 0 (interrupts enabled), unless you call it directly first.

If you don’t directly invoke `nanospin_calibrate()`, the first `nanospin*()` call in a process will have an overly long delay.

The `nanospin*()` functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX `timer_*()` functions.

Interrupts occurring during `nanospin_calibrate()` can throw off its timings. If `disable` is 0 (zero), you can prevent this situation by:

1. letting the thread acquire I/O privilege
2. disabling the interrupts around the `nanospin_calibrate()` call.
If `disable` is 1 (one), the code disables interrupts around the calibration loop(s). The calling thread is still responsible for obtaining I/O privilege before calling `nanospin_calibrate()`.

**Returns:**

- **EOK** Success.
- **EINTR** A too-high rate of interrupts occurred during the calibration routine.
- **EPERM** The process doesn’t have superuser capabilities.

**Examples:**

Busy-wait for 100 nanoseconds:

```c
#include <time.h>
#include <sys/syspage.h>

int disable = 0;
unsigned long time = 100;
...

/* Wake up the hardware, then wait for it to be ready. */
if ( (nanospin_calibrate( disable )) == EOK )
    nanospin_count( nanospin_ns_to_count( time ) );
else
    printf ("Didn’t calibrate successfully.\n");

/* Use the hardware. */
...
```

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No

    continued…
**nanospin_calibrate()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

nanospin(), nanospin_count(), nanospin_ns(), nanospin_ns_to_count(), timer_create()
**nanospin_count()**

Busy-wait without blocking for a number of iterations

### Synopsis:

```c
#include <time.h>

void nanospin_count( unsigned long count );
```

### Arguments:

- `count`  
  The number of iterations that you want to busy-wait for.

### Library:

- **libc**
  Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `nanospin_count()` function busy-waits for the number of iterations specified in `count`. Use `nanospin_ns_to_count()` to turn a number of nanoseconds into an iteration count suitable for `nanospin_count()`.

The `nanospin*()` functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX `timer_*()` functions.

### Examples:

Busy-wait for at least 100 nanoseconds:

```c
#include <time.h>
#include <sys/syspage.h>

unsigned long time = 100;
...

/* Wake up the hardware, then wait for it to be ready. */

nanospin_count( nanospin_ns_to_count( time ) );
```
/* Use the hardware. */
...

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

You should use busy-waiting only when absolutely necessary for accessing hardware.

See also:

nanosleep(), nanospin(), nanospin_calibrate(), nanospin_ns(), nanospin_ns_to_count(), sched_yield(), sleep(), timer_create()
Synopsis:

```c
#include <time.h>

int nanospin_ns( unsigned long nsec );
```

Arguments:

**nsec** The number of nanoseconds that you want to busy-wait for.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `nanospin_ns()` function busy-waits for the number of nanoseconds specified in **nsec**, without blocking the calling thread.

The `nanospin*()` functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX `timer_*()` functions.

The first time you call `nanospin_ns()`, the C library invokes `nanospin_calibrate()` with an argument of 0 (interrupts enabled), if you haven’t invoked it directly first.

Returns:

**EOK** Success.

**E2BIG** The delay specified by **nsec** is greater than 500 milliseconds.
nanospin_ns()

A delay of more than a few milliseconds might not work on some processors. For longer delays, use the POSIX `timer_*()` functions.

EINTR A too-high rate of interrupts occurred during the calibration routine.

ENOSYS This system’s `startup-*` program didn’t initialize the timing information necessary to use `nanospin_ns()`.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

You should use busy-waiting only when absolutely necessary for accessing hardware.

It isn’t safe to call this function in an interrupt handler if `nanospin_calibrate()` hasn’t been called yet.

**See also:**

`nanosleep()`, `nanospin()`, `nanospin_calibrate()`, `nanospin_count()`, `nanospin_ns_to_count()`, `sched_yield()`, `sleep()`, `timer_create()`
nanospin ns_to_count()

Convert a time in nanoseconds into a number of iterations

Synopsis:

```c
#include <time.h>

unsigned long nanospin_ns_to_count(
    unsigned long nsec );
```

Arguments:

nsec   The number of nanoseconds that you want to convert.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `nanospin_ns_to_count()` function converts the number of nanoseconds specified in `nsec` into an iteration count suitable for `nanospin_count()`.

The `nanospin*()` functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX `timer_*()` functions.

The first time that you call `nanospin_ns_to_count()`, the C library invokes `nanospin_calibrate()` with an argument of 0 (interrupts enabled), if you haven’t invoked it directly first.

Returns:

The amount of time to busy-wait, or -1 if an error occurred (`errno` is set).
**nanospin_ns_to_count()**

**Errors:**

- **EINVAL**  A too-high rate of interrupts occurred during the calibration routine.
- **ENOSYS** This system’s **startup-** program didn’t initialize the timing information necessary to use **nanospin_ns_to_count().**

**Examples:**

Busy-wait for at least one nanosecond:

```c
#include <time.h>
#include <sys/syspage.h>

unsigned long time = 1;
...

/* Wake up the hardware, then wait for it to be ready. */

/*
 * The C library invokes nanospin_calibrate if it hasn’t already been called.
 */

nanospin_count( nanospin_ns_to_count( time ) );

/* Use the hardware. */
...
```

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: Read the **Caveats**
- Signal handler: Yes
- Thread: Yes
nanospin_ns_to_count()

Caveats:

You should use busy-waiting only when absolutely necessary for accessing hardware.

It isn’t safe to call this function in an interrupt handler if nanospin_calibrate() hasn’t been called yet.

See also:

nanospin(), nanospin_calibrate(), nanospin_count(), nanospin_ns(), timer_create()
**nap()**

Sleep for a given number of milliseconds

**Synopsis:**

```c
#include <unix.h>

unsigned int nap( unsigned int ms);
```

**Arguments:**

- `ms` The number of milliseconds that you want the process to sleep.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `nap()` routine delays the calling process for `ms` milliseconds. This function is the same as `delay()` and is similar to `napms()`.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`delay()`, `napms()`
Synopsis:

```
#include <curses.h>

int napms(int ms);
```

Arguments:

- `ms` The number of milliseconds that you want the process to sleep.

Library:

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `napms()` routine delays the calling process for `ms` milliseconds. This function is similar to `delay()` and `nap()`.

Classification:

- Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`delay()`, `nap()`
nbaco\texttt{connect()}

\begin{verbatim}
#include <sys/socket.h>

int nbaco\texttt{connect}( int \texttt{s},
    const struct sockaddr * \texttt{name},
    size_t \texttt{namelen} );
\end{verbatim}

\textbf{Arguments:}

\begin{itemize}
    \item \texttt{s}  The descriptor of the socket on which to initiate the connection.
    \item \texttt{name}  The name of the socket to connect to for a \texttt{SOCK\textunderscore STREAM} connection.
    \item \texttt{namelen}  The length of the \texttt{name}, in bytes.
\end{itemize}

\textbf{Library:}

\texttt{libsocket}

Use the \texttt{-l socket} option to \texttt{qcc} to link against this library.

\textbf{Description:}

The \texttt{nbaco\texttt{connect}()} function is called in place of \texttt{connect()}, to prevent a nonblocking \texttt{connect()} from blocking during an autoconnect (see \texttt{/etc/autoconnect}).

When the autoconnect behavior is used, \texttt{connect()} may block your application while waiting for the autoconnect to complete; \texttt{nbaco\texttt{connect}()} allows your application to continue executing during the autoconnect.

The \texttt{nbaco\texttt{connect}()} function takes the same arguments as \texttt{connect()}, but it differs in the return value when an autoconnect is required. If an autoconnect is required, a file descriptor (\texttt{fd}) is returned. The \texttt{fd} is used in the call to \texttt{nbaco\texttt{connect\_result}()} to get the \texttt{errno} related to the autoconnect and the connect attempt.
Since \texttt{nbaco\textunderscore connect\_result()} is a blocking call, it’s recommended that you call \texttt{select()} first to determine if there’s data available on the pipe.

When the data’s available, call \texttt{nbaco\textunderscore connect\_result()} to get the status of the \texttt{nbaco\textunderscore connect()} attempt.

If an autoconnect isn’t required, \texttt{nbaco\textunderscore connect()} returns -1 and exhibits the same behavior as \texttt{connect()} on nonblocking sockets (e.g. if -1 is returned and \texttt{errno} is set to EINPROGRESS, it’s possible to do a \texttt{select()} for completion by selecting the socket for writing).

**Returns:**

A file descriptor that you can pass to \texttt{nbaco\textunderscore connect\_result()} to get the result of the \texttt{nbaco\textunderscore connect()} attempt, or -1 if an error occurred (\texttt{errno} is set).

**Errors:**

Any value from the Errors section in \texttt{connect()}, as well as:

\begin{itemize}
  \item EINVAL The socket file descriptor being passed isn’t nonblocking.
\end{itemize}

**Classification:**

QNX Neutrino

\begin{table}[H]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\hline
\end{tabular}
\end{table}
Caveats:

The `pipe` manager must be available.

See also:

`accept()`, `bind()`, `connect()`, `errno`, `fcntl()`, `getsockname()`,
`nbacnect_result()`, `open()`, `pipe()`, `read()`, `select()`, `socket()`, `write()`

`/etc/autoconnect.pipe` in the Utilities Reference.
**Synopsis:**

```c
#include <sys/socket.h>

int nbaconnect_result( int fd,
                      int * error );
```

**Arguments:**

- `fd` The file descriptor returned by `nbaconnect()`.
- `error` A pointer to a location where the function can store the status.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `nbaconnect_result()` function gets the status of the previous `nbaconnect()` call when an `fd` was returned. Since `nbaconnect_result()` is a blocking call, it’s best to test the status of the `fd` with a call to `select()` to verify that the file descriptor is ready to be read.

When there’s data available, the status is put in `error`, which may be any of the `errno` values set by `connect()` during an attempt to make a non-blocking connection.

The `fd` is always closed by this function whether or not there’s a status to report.

**Returns:**

- `0` The call was successful; `error` contains the status.
- `-1` An error occurred while obtaining the status.
Errors:

Any value from the Errors section in `connect()`, as well as:
- EBADF Invalid fd.
- ENOMSG There’s no data, or not enough data, from the fd.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`connect()`, `nbacconnect()`, `select()`

`autoconnect` in the Utilities Reference.
Synopsis:
```c
#include <sys/netmgr.h>

#define ND_NODE_CMP(a, b) ...
```

Arguments:
- `a`, `b` The node descriptors that you want to compare. You can use either the value 0 or `ND_LOCAL_NODE` to refer to the local node.

Library:
- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `ND_NODE_CMP()` macro compares two node descriptors.

Returns:
- `< 0` The node descriptor `a` is less than `b`.
- `0` The descriptors refer to the same machine.
- `> 0` The node descriptor `a` is greater than `b`.

Examples:
```c
#include <sys/neutrino.h>

uint32_t nd1, nd2;

if ( ND_NODE_CMP(nd1, nd2) == 0 ) {
  /* Same node */
  ...
} else {
  /* Different nodes */
  ...
}
```
### Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

- `netmgr_ntodtostr()`, `netmgr_remote_nd()`, `netmgr_strtond()`
- Qnet Networking chapter of the *Programmer’s Guide*
- Qnet Networking chapter of the *System Architecture* guide
Structure for information from the network database

Synopsis:

```c
#include <netdb.h>

struct netent {
    char * n_name;
    char ** n_aliases;
    int n_addrtype;
    uint32_t n_net;
};
```

Description:

This structure holds information from the network database, `/etc/networks`.

The members of this structure are:

- **n_name**: The name of the network.
- **n_aliases**: A zero-terminated list of alternate names for the network.
- **n_addrtype**: The type of the network number returned; currently only AF_INET.
- **n_net**: The network number. Network numbers are returned in machine-byte order.

Classification:

POSIX 1003.1

See also:

- `endnetent()`, `getnetbyaddr()`, `getnetbyname()`, `getnetent()`, `setnetent()`
- `/etc/networks` in the Utilities Reference
**Synopsis:**

```c
#include <sys/netmgr.h>

int netmgr_ndtostr( unsigned flags,
                    int nd,
                    char * buf,
                    size_t maxbuf );
```

**Arguments:**

- `flags` Which part(s) of the Fully Qualified Path Name (FQPN) to adjust; see below.
- `nd` The node descriptor that you want to convert.
- `buf` A pointer to a buffer where the function can store the converted identifier.
- `maxbuf` The size of the buffer.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `netmgr_ndtostr()` function converts a node descriptor, `nd`, to a string and stores the string in the buffer pointed to by `buf`. The size of the buffer is given by `maxbuf`.

A node descriptor is a temporary numeric description of a remote node. For more information, see the Qnet Networking chapter of the `System Architecture` guide.

The `flags` argument indicates which part(s) of the Fully Qualified Path Name (FQPN) to adjust. The following diagram shows the components for the FQPN, `/net/qnet.qnx.com~preferred:`
netmgr_ndtostr()

Components of a fully qualified pathname.

The Fully Qualified Node Name (FQNN) is `qnet.qnx.com`.

The default string that `netmgr_ndtostr()` builds is the FQNN, plus the Quality of Service (QoS) if it isn’t the default (`~loadbalance`). You can pass this string to any other node that can call `netmgr_strtond()` and has a `nd` that refers to the same node with the same QoS.

A bitwise OR of flags modify the default string in the following way:

**ND2S_DIR_HIDE**

Never include the directory.

**ND2S_DIR_SHOW**

Always build a pathname to the root directory (i.e. `/`) of the node indicated by `nd`. For example, calling:

```c
netmgr_ndtostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)
```

on a node with a default domain of `qnx.com` using a `nd` that refers to a FQNN of `peterv.qnx.com` results in the string, `/net/peterv.qnx.com/`.

If Qnet isn’t active on the node, and `netmgr_ndtostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)` has a `nd` of `ND_LOCAL_NODE`, then the string is `/`.

**ND2S_DOMAIN_HIDE**

Never include the domain.

**ND2S_DOMAIN_SHOW**

Always include the domain.
ND2S_LOCAL_STR
Display shorter node names on your local node. For example, calling:

```c
netmgr_ndtostr(ND2S_LOCAL_STR, nd, buf, sizeofbuf)
```

on a node with a default domain of `qnx.com` using a `nd` that refers to an FQPN of `/net/peterv.qnx.com` results in a string of `peterv`.

Whereas a `nd` that refers to `/net/peterv.anotherhost.com` results in `peterv.anotherhost.com`.

ND2S_NAME_HIDE
Never include the name.

ND2S_NAME_SHOW
Always include the name.

ND2S_QOS_HIDE
Never include the quality of service (QoS).

ND2S_QOS_SHOW
Always include the QoS.

ND2S_SEP_FORCE
Always include a leading separator. For example, calling:

```c
netmgr_ndtostr(ND2S_SEP_FORCE | ND2S_DIR_HIDE | ND2S_QOS_SHOW, nd, buf, sizeofbuf)
```

with a `nd` of ND_LOCAL_NODE results in a string of `~loadbalance`. This is useful if you want to concatenate each component of the FQPN individually.
Don’t use a ND2S_*_HIDE and a corresponding ND2S_*_SHOW together.

Returns:

The length of the string, or -1 if an error occurs (errno is set).

Errors:

ENOTSUP Qnet isn’t running.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/netmgr.h>

int main ()
{
    int nd1, nd2, nd3, len;
    char path1[50] = "/net/dave",
            path2[50] = "/net/karen",
            buff[100];

    nd1 = netmgr_strtond( path1, NULL);
    if (nd1 == -1) {
        perror ("netmgr_strtond" );
        return EXIT_FAILURE;
    } else {
        printf ("Node id for %s is %d\n", path1, nd1);
    }

    nd2 = netmgr_strtond( path2, NULL);
    if (nd2 == -1) {
        perror ("netmgr_strtond" );
        return EXIT_FAILURE;
    } else {
        printf ("Node id for %s is %d\n", path2, nd2);
    }

    nd3 = netmgr_remote_nd ( nd2, nd1 );
    if (nd3 == -1) {
        perror ("netmgr_strtond" );
        return EXIT_FAILURE;
    }
}
```
null

Classification:

QNX Neutrino

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

See also:

ND_NODE_CMP(), netmgr_remote_nd(), netmgr_strtond()

Qnet Networking chapter of the Programmer’s Guide

Qnet Networking chapter of the System Architecture guide
netmgr_remote_nd()

Get a node descriptor that's relative to a remote node

Synopsis:

```c
#include <sys/netmgr.h>

int netmgr_remote_nd( int remote_nd,
                        int local_nd );
```

Arguments:

- `remote_nd` The node descriptor of a remote node.
- `local_nd` A node descriptor, relative to the local node, that you want to convert to be relative to the remote node.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `netmgr_remote_nd()` function converts a node descriptor that’s relative to the local node into a node descriptor that’s relative to the specified remote node.

Returns:

The node descriptor, relative to the remote node, or -1 if an error occurred (`errno` is set).

Examples:

See `netmgr_ndtostr()`.

Classification:

QNX Neutrino
### netmgr_remote_nd()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `ND_NODE_CMP()`, `netmgr_ndtostr()`, `netmgr_strtond()`
- Qnet Networking chapter of the *Programmer’s Guide*
- Qnet Networking chapter of the *System Architecture* guide
netmgr_strtond()

Convert a string into a node descriptor

Synopsis:

```c
#include <sys/netmgr.h>

int netmgr_strtond( const char * nodename,
                     char ** endstr );
```

Arguments:

- `nodename`: The string that you want to convert into a node descriptor.
- `endstr`: NULL, or the address of a location where the function can store a pointer to the character after the node name in the string.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `netmgr_strtond()` function converts a string to a node descriptor. If `endstr` isn’t NULL, it’s set to point to the character after the node name in the given string.

Returns:

The node descriptor, or -1 if an error occurred (`errno` is set).

Errors:

- `ENOTSUP`: Qnet isn’t running.
Examples:

See netmgr_ndtost().

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ND_NODE_CMP(), netmgr_ndtost(), netmgr_remote_nd()

Qnet Networking chapter of the Programmer’s Guide

Qnet Networking chapter of the System Architecture guide
nextafter(), nextafterf()

Compute the next representable double-precision floating-point number

Synopsis:

```c
#include <math.h>

double nextafter ( double x,
                    double y);

float nextafterf ( float x,
                    float y);
```

Arguments:

- `x`  The number that you want the next number after.
- `y`  A number that specifies the direction you want to go; see below.

Library:

```c
libm
```

Use the `-l m` option to qcc to link against this library.

Description:

The `nextafter()` and `nextafterf()` functions compute the next representable double-precision floating-point value following `x` in the direction of `y`.

Returns:

The next machine floating-point number of `x` in the direction towards `y`.

If: `nextafter()` returns:

- `y < x`  The next possible floating-point value less than `y`
- `y > x`  The next possible floating-point value greater than `x`

continued…
**nextafter(), nextafterf()**

© 2007, QNX Software Systems GmbH & Co. KG.

If:  

<table>
<thead>
<tr>
<th>nextafter() returns:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ is NAN</td>
<td>NAN</td>
</tr>
<tr>
<td>$y$ is NAN</td>
<td>NAN</td>
</tr>
<tr>
<td>$x$ is finite</td>
<td>±HUGE_VAL, according to the sign of $x$ ($errno$ is set)</td>
</tr>
</tbody>
</table>

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set $errno$ to 0, call the function, and then check $errno$ again. These functions don’t change $errno$ if no errors occurred.

**Examples:**
```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

void dump_to_hex(double d) {
    printf("0x%08x %08x \n",
            (*((uint64_t*)&d) >> 32),
            (*((uint64_t*)&d) & 0xffffffff));
}

int main(int argc, char** argv)
{
    double a, b, c;
    a = 0;
    b = nextafter(a, -1);  
    c = nextafter(a, 1);  
    printf("Next possible value before %f is %f \n", a, b);  
    printf("-->") ; dump_to_hex(a);  
    printf("-->") ; dump_to_hex(b);  
    printf("Next possible value after %f is %f \n", a, c);  
    printf("-->") ; dump_to_hex(a);  
    printf("-->") ; dump_to_hex(c);  
    return(0);
}
```

produces the output:

1928
nextafter(), nextafterf()

Next possible value before 0.000000 is 0.000000
-->0x00000000 00000000
-->0x80000000 00000001
Next possible value after 0.000000 is 0.000000
-->0x00000000 00000000

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**nftw(), nftw64()**

Walk a file tree

### Synopsis:

```c
#include <ftw.h>

int nftw( const char *path,
           int (*fn)( const char *fname,
                     const struct stat *sbuf,
                     int flags,
                     struct FTW *ftw),
           int depth,
           int flags);
```

### Arguments:

- **path**
  The path of the directory whose file tree you want to walk.

- **fn**
  A pointer to a function that you want to call for each file; see below.

- **depth**
  The maximum number of file descriptors that `nftw()` can use. The `nftw()` function uses one file descriptor for each level in the tree.
  If `depth` is zero or negative, the effect is the same as if it were 1. The `depth` must not be greater than the number of file descriptors currently available for use. The `nftw()` function is faster if `depth` is at least as large as the number of levels in the tree.

- **flags**
  The value of `flags` is constructed by the bitwise ORing of values from the following list, defined in the `<ftw.h>` header file.

  - **FTW_CHDIR**
    If set, `nftw()` changes the current working directory to each directory as it reports files in that directory.

  - **FTW_DEPTH**
    If set, `nftw()` reports all files in a directory before reporting the directory itself (otherwise the directory is reported before any file it contains).
FTW_MOUNT  If set, nftw() only reports files on the same filesystem as path.

FTW_PHYS   If set, nftw() performs a physical walk and doesn’t follow any symbolic link.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

These functions are in libc.a, but not in libc.so (in order to save space).

Description:

The nftw() function recursively descends the directory hierarchy identified by path. For each object in the hierarchy, nftw() calls the user-defined function fn(), passing to it:

- a pointer to a NULL-terminated character string containing the name of the object
- a pointer to a stat structure (see stat()) containing information about the object
- an integer. Possible values of the integer, defined in the <nftw.h> header, are:

  FTW_F  The object is a file.
  FTW_D  The object is a directory.
  FTW_DNR The object is a directory that can’t be read. Descendents of the directory aren’t processed.
  FTW_DP  The object is a directory, and its contents have been reported. See the FTW_DEPTH flag above.
FTW_NS    The `stat()` failed on the object because the permissions weren’t appropriate. The `stat` buffer passed to `fn()` is undefined.

FTW_SL    The object is a symbolic link. See the FTW_PHYS flag above.

FTW_SLN   The object is a symbolic link that does not name an existing file.

- a pointer to a FTW structure, which contains the following fields:

  base     The offset of the objects filename in the pathname passed as the first argument to `fn()`.

  level    The depth relative to the root of the walk (where the root is level 0).

  quit     A flag that can be set to control the behaviour of `nftw()` within the current directory. If assigned, it may be given the following values:

          FTW_SKR Skip the remainder of this directory
          FTW_SKD If the object is FTW_D, then do not enter into this directory.

The tree traversal continues until the tree is exhausted, an invocation of `fn()` returns a nonzero value, or some error is detected within `nftw()` (such as an I/O error). If the tree is exhausted, `nftw()` returns zero. If `fn()` returns a nonzero value, `nftw()` stops its tree traversal and returns whatever value was returned by `fn()`.

When `nftw()` returns, it closes any file descriptors it opened; it doesn’t close any file descriptors that may have been opened by `fn()`.

**Returns:**

- 0     Success.

- -1    An error (other than EACCESS) occurred (`errno` is set).
Classification:

*nftw()* is POSIX 1003.1 XSI; *nftw64()* is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Because *nftw()* is recursive, it might terminate with a memory fault when applied to very deep file structures.

This function uses *malloc()* to allocate dynamic storage during its operation. If *nftw()* is forcibly terminated, for example if *longjmp()* is executed by *fn1()* or an interrupt routine, *nftw()* doesn’t have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn1()* return a nonzero value at its next invocation.

See also:

*ftw(), longjmp(), malloc(), stat*
nice()
Change the priority of a process

Synopsis:
#include <unistd.h>

int nice( int incr );

Arguments:
incr    The amount that you want to add to the process’s priority.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The nice() function allows a process to change its priority. The invoking process must be in a scheduling class that supports the operation.

The nice() function adds the value of incr to the nice value of the calling process. A process’s nice value is a nonnegative number; a greater positive value results in a lower CPU priority.

A maximum nice value of 2 * NZERO - 1 and a minimum nice value of 0 are imposed by the system. NZERO is defined in <limits.h> with a default value of 20. If you request a value above or below these limits, the nice value is set to the corresponding limit. A nice value of 40 is treated as 39. Only a process with superuser privileges can lower the nice value.

Returns:
The new nice value minus NZERO. If an error occurred, -1 is returned, the process’s nice value isn’t changed, and errno is set.
Errors:

- **EINVAL**: The `nice()` function was called by a process in a scheduling class other than time-sharing.
- **EPERM**: The `incr` argument was negative or greater than 40, and the effective user ID of the calling process isn’t the superuser.

Classification:

- **POSIX 1003.1 XSI**

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

As -1 is a permissible return value in a successful situation, an application wishing to check for error situations should set `errno` to 0, then call `nice()`, and if it returns -1, check to see if `errno` is nonzero.

See also:

`excl()`, `excelet()`, `exclp()`, `excelep()`, `execv()`, `execve()`, `execvp()`, `execvpe()`,
`nice` in the Utilities Reference
nrand48()  
© 2007, QNX Software Systems GmbH & Co. KG.

Generate a pseudo-random nonnegative long integer in a thread-safe manner

Synopsis:
```c
#include <stdlib.h>

long nrand48( unsigned short xsubi[3] );
```

Arguments:
- `xsubi`: An array that comprises the 48 bits of the initial value that you want to use.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `nrand48()` function uses a linear congruential algorithm and 48-bit integer arithmetic to generate a nonnegative long integer uniformly distributed over the interval \([0, 2^{31}]\).

The `xsubi` array should contain the desired initial value; this makes `nrand48()` thread-safe, and lets you start a sequence of random numbers at any known value.

Returns:
A pseudo-random long integer.

Classification:
POSIX 1003.1 XSI

Safety
- Cancellation point: No
  - continued…

1936  C Library — M to O  September 10, 2007
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `mrand48()`, `seed48()`, `srand48()`
Convert nanoseconds to a `timespec` structure

Synopsis:

```c
#include <time.h>

void nsec2timespec( struct timespec *timespec_p,
                    _uint64 nsec );
```

Arguments:

- `timespec_p` A pointer to a `timespec` structure where the function can store the converted time.
- `nsec` The number of nanoseconds that you want to convert.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This function converts the given number of nanoseconds, `nsec`, into seconds and nanoseconds, and stores them in the `timespec` structure pointed to by `timespec_p`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`timespec`, `timespec2nsec()`
ntohl()  
Convert a 32-bit value from network-byte order to host-byte order

Synopsis:
#include <arpa/inet.h>

uint32_t ntohl( uint32_t netlong );

Arguments:
netlong  The value that you want to convert.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The ntohs() function converts a 32-bit value from network-byte order to host-byte order. If a machine’s byte order is the same as the network order, this routine is defined as a null macro.
You most often use this routine in conjunction with internet addresses and ports returned by gethostbyname() and getservent().

Returns:
The value in host-byte order.

Classification:
POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
Safety

| Thread | Yes |

See also:

gethostbyname(), getservent(), htonl(), htons(), ntohs()
ntohs()  © 2007, QNX Software Systems GmbH & Co. KG.

Convert a 16-bit value from network-byte order to host-byte order

Synopsis:

```
#include <arpa/inet.h>

uint16_t ntohs( uint16_t netshort );
```

Arguments:

```
netshort   The value that you want to convert.
```

Library:

```
libc
```

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `ntohs()` function converts a 16-bit value from network-byte order to host-byte order. If a machine’s byte order is the same as the network order, this routine is defined as a null macro.

You most often use this routine in conjunction with internet addresses and ports returned by `gethostbyname()` and `getservent()`.

Returns:

The value in host-byte order.

Classification:

```
POSIX 1003.1
```

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
See also:

`gethostbyname()`, `getservent()`, `htonl()`, `htons()`, `ntohl()`
offsetof()  
© 2007, QNX Software Systems GmbH & Co. KG.
Return the offset of an element within a structure

Synopsis:
```c
#include <stddef.h>

#define offsetof(composite, name) ...
```

Arguments:
- `composite` A `struct` or `union`.
- `name` The name of an element in `composite`.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `offsetof()` macro returns the offset of the element `name` within the `struct` or `union` `composite`.
This provides a portable method to determine the offset.

Returns:
The offset of `name`.

Examples:
```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

struct new_def
{
    char *first;
    char second[10];
    int third;
};

int main( void )
{
```
```c
printf( "first:%d second:%d third:%d\n",
    offsetof( struct new_def, first ),
    offsetof( struct new_def, second ),
    offsetof( struct new_def, third ) );

return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

This is a macro.
open(), open64()

Open a file

Synopsis:

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open( const char * path,
           int oflag,
           ... );

int open64( const char * path,
            int oflag,
            ... );
```

Arguments:

- `path` The path name of the file that you want to open.
- `oflag` Flags that specify the status and access modes of the file; see below.

If you set `O_CREAT` in `oflag`, you must also specify the following argument:

- `mode` An object of type `mode_t` that specifies the access mode that you want to use for a newly created file. For more information, see “Access permissions” in the documentation for `stat()`, and the description of `O_CREAT`, below.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
open(), open64()

Description:
The open() and open64() functions open the file named by path, creating an open file description that refers to the file, and a file descriptor that refers to the file description. The file status flags and the file access modes of the open file description are set according to the value of oflag.

These functions ignore any advisory locks that you set with fcntl().

The open file descriptor created is new, and therefore isn’t shared with any other process in the system.

Construct the value of oflag by bitwise ORing values from the following list, defined in the <fcntl.h> header file. You must specify exactly one of the first three values (file access modes) below in the value of oflag:

- O_RDONLY Open for reading only.
- O_RDWR Open for reading and writing. Opening a FIFO for read-write is unsupported.
- O_WRONLY Open for writing only.

You can also specify any combination of the remaining flags in the value of oflag:

- O_APPEND If set, the file offset is set to the end of the file prior to each write.
- O_CLOEXEC Close the file descriptor on execution.
- O_CREAT This option requires a third argument, mode, which is of type mode_t. If the file exists, this flag has no effect, except in combination with O_EXCL as noted below.

Otherwise, the file is created; the file’s user ID is set to the effective user ID of the process; the
group ID is set to the effective group ID of the process or the group ID of the file’s parent directory (see `chmod()`).

The permission bits, as defined in `<sys/stat.h>`, are set to the value of `mode`, except those bits set in the process’s file mode creation mask (see `umask()` for details). Bits set in `mode` other than the file permission bits (i.e. the file type bits) are ignored. The `mode` argument doesn’t affect whether the file is opened for reading, for writing, or for both.

**O_DSYNC**

If set, this flag affects subsequent I/O calls; each call to `write()` waits until all data is successfully transferred to the storage device such that it’s readable on any subsequent open of the file (even one that follows a system failure) in the absence of a failure of the physical storage medium. If the physical storage medium implements a non-write-through cache, then a system failure may be interpreted as a failure of the physical storage medium, and data may not be readable even if this flag is set and the `write()` indicates that it succeeded.

**O_EXCL**

If you set both `O_EXCL` and `O_CREAT`, `open()` fails if the file exists. The check for the existence of the file and the creation of the file if it doesn’t exist are atomic; no other process that’s attempting the same operation with the same filename at the same time will succeed. Specifying `O_EXCL` without `O_CREAT` has no effect.

**O_LARGEFILE**

Allow the file offset to be 64 bits long.

**O_NOCTTY**

If set, and `path` identifies a terminal device, the `open()` function doesn’t cause the terminal device to become the controlling terminal for the process.
O_NONBLOCK

- When opening a FIFO with O_RDONLY or O_WRONLY set:
  
  If O_NONBLOCK is set:
  
  Calling open() for reading-only returns without delay. Calling open() for writing-only returns an error if no process currently has the FIFO open for reading.

  If O_NONBLOCK is clear:
  
  Calling open() for reading-only blocks until a process opens the file for writing. Calling open() for writing-only blocks until a process opens the file for reading.

- When opening a block special or character special file that supports nonblocking opens:
  
  If O_NONBLOCK is set:
  
  The open() function returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

  If O_NONBLOCK is clear:
  
  The open() function waits until the device is ready or available before returning. The definition of when a device is ready is device-specific.

- Otherwise, the behavior of O_NONBLOCK is unspecified.

O_REALIDS

Use the real uid/gid for permissions checking.

O_RSYNC

Read I/O operations on the file descriptor complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags.

O_SYNC

If set, this flag affects subsequent I/O calls; each call to read() or write() is complete only when both the data has been successfully transferred (either
read or written) and all file system information relevant to that I/O operation (including that required to retrieve said data) is successfully transferred, including file update and/or access times, and so on. See the discussion of a successful data transfer in O_DSYNC, above.

O_TRUNC

If the file exists and is a regular file, and the file is successfully opened O_WRONLY or O_RDWR, the file length is truncated to zero and the mode and owner are left unchanged. O_TRUNC has no effect on FIFO or block or character special files or directories. Using O_TRUNC with O_RDONLY has no effect.

The largest value that can be represented correctly in an object of type off_t shall be established as the offset maximum in the open file description.

Returns:

A nonnegative integer representing the lowest numbered unused file descriptor. On a file capable of seeking, the file offset is set to the beginning of the file. Otherwise, -1 is returned (errno is set).

In QNX Neutrino, the returned file descriptor is the same as the connection ID (or coid) used by the Neutrino-specific functions.

Errors:

EACCES

Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by oflag are denied, or the file doesn’t exist and write permission is denied for the parent directory of the file to be created.

EBADFSYS

While attempting to open the named file, either the file itself or a component of the path prefix was
found to be corrupted. A system failure — from which no automatic recovery is possible — occurred while the file was being written to, or while the directory was being updated. You’ll need to invoke appropriate systems-administration procedures to correct this situation before proceeding.

EBUSY
File access was denied due to a conflicting open (see sopen()).

EEXIST
The O_CREAT and O_EXCL flags are set, and the named file exists.

EINTR
The open() operation was interrupted by a signal.

EINVAL
The requested synchronized modes (O_SYNC, O_DSYNC, O_RSYNC) aren’t supported.

EISDIR
The named file is a directory, and the oflag argument specifies write-only or read/write access.

ELOOP
Too many levels of symbolic links or prefixes.

EMFILE
Too many file descriptors are currently in use by this process.

ENAMETOOLONG
The length of the path string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.

ENFILE
Too many files are currently open in the system.

ENOENT
The O_CREAT flag isn’t set, and the named file doesn’t exist; or O_CREAT is set and either the path prefix doesn’t exist, or the path argument points to an empty string.

ENOSPC
The directory or filesystem that would contain the new file can’t be extended.
ENOSYS The `open()` function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of the path prefix isn’t a directory.

ENXIO The O_NONBLOCK flag is set, the named file is a FIFO, O_WRONLY is set, no process has the file open for reading, or the media associated with the file has been removed (e.g. CD, floppy).

EOVERFLOW The named file is a regular file and the size of the file can’t be represented correctly in an object of type `off_t`.

ERofs The named file resides on a read-only filesystem and either O_WRONLY, O_RDWR, O_CREAT (if the file doesn’t exist), or O_TRUNC is set in the oflag argument.

Examples:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main( void )
{
  int fd;

  /* open a file for output */
  /* replace existing file if it exists */
  /* with read/write perms for owner */
  fd = open( "myfile.dat", O_WRONLY | O_CREAT | O_TRUNC, 
             S_IRUSR | S_IWUSR );

  /* read a file that is assumed to exist */
  fd = open( "myfile.dat", O_RDONLY );

  /* append to the end of an existing file */
  /* write a new file if file doesn’t exist */
  /* with full read/write permissions */
```
open(), open64()

```c
fd = open( "myfile.dat",
    O_WRONLY | O_CREAT | O_APPEND,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH );
return EXIT_SUCCESS;
}
```

**Classification:**

`open()` is POSIX 1003.1; `open64()` is Large-file support

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The `open()` function includes POSIX 1003.1-1996 and QNX extensions.

**See also:**

`chmod()`, `close()`, `creat()`, `dup()`, `dup2()`, `errno`, `fcntl()`, `fstat()`, `lseek()`, `read()`, `write()`
opendir()  
Open a directory

Synopsis:

```c
#include <dirent.h>

DIR * opendir( const char * dirname );
```

Arguments:

dirname The path of the directory to be opened. It can be relative to the current working directory, or an absolute path.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The opendir() function is used with readdir() and closedir() to get the list of file names contained in the directory specified by dirname.

You can read more than one directory at the same time using the opendir(), readdir(), rewinddir() and closedir() functions.

The result of using a directory stream after one of the exec*() or spawn*() functions is undefined. After a call to the fork() function, either the parent or the child (but not both) can continue processing the directory stream using readdir() and rewinddir(). If both the parent and child processes use these functions, the result is undefined. Either process can use closedir().

Returns:

A pointer to a DIR structure required for subsequent calls to readdir() to retrieve the file names in dirname, or NULL if dirname isn’t a valid path (errno is set).
Errors:

- **EACCES**: Search permission is denied for a component of `dirname`, or read permission is denied for `dirname`.
- **ELOOP**: Too many levels of symbolic links or prefixes.
- **ENAMETOOLONG**: The length of `dirname` exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
- **ENOENT**: The named directory doesn’t exist.
- **ENOSYS**: The `opendir()` function isn’t implemented for the filesystem specified in `dirname`.
- **ENOTDIR**: A component of `dirname` isn’t a directory.

Examples:

Get a list of files contained in the directory `/home/fred`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main( void )
{
    DIR* dirp;
    struct dirent* direntp;
    dirp = opendir( "./home/fred" );
    if( dirp == NULL ) {
        perror( "can't open /home/fred" );
    } else {
        direntp = readdir( dirp );
        if( direntp == NULL ) break;
        printf( "%s\n", direntp->d_name );
    }
    closedir( dirp );
    return EXIT_SUCCESS;
}
```
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

closedir(), errno, readdir(), readdir_r(), rewinddir(), seekdir(), telldir()
openfd()

Open for private access a file associated with a given descriptor

Synopsis:

```c
#include <unistd.h>

int openfd( int fd, int oflag );
```

Arguments:

- `fd` A file descriptor associated with the file that you want to open.
- `oflag` How you want to open the file; a combination of the following bits:
  - `O_RDONLY` — permit the file to be only read.
  - `O_WRONLY` — permit the file to be only written.
  - `O_RDWR` — permit the file to be both read and written.
  - `O_APPEND` — cause each record that’s written to be written at the end of the file.
  - `O_TRUNC` — truncate the file to contain no data.

Library:

`libc`

Use the `l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `openfd()` function opens the file associated with the file descriptor, `fd`. This is similar to `dup()`, except the new `fd` has private access modes and offset. The access mode, `oflag`, must be equal to or more restrictive than the access mode of the source `fd`. 
openfd()

Returns:
A file descriptor, or -1 if an error occurred (errno is set).

Errors:

- EBADF  Invalid file descriptor fd.
- EACCES The access mode specified by oflag isn’t equal to or more restrictive than the access mode of the source fd.
- EBUSY  Sharing mode (sflag) was denied due to a conflicting open (see sopenfd()).

Examples:
```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main ( void )
{
    int fd, fd2, oflag;

    fd = open ("etc/passwd", O_RDONLY);
    fd2 = openfd ( fd, O_RDONLY );
    return EXIT_SUCCESS;
}
```

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1958   C Library — M to O   September 10, 2007
See also:

`dup()`, `sopenfd()`
openlog()
Open the system log

Synopsis:

```c
#include <syslog.h>

void openlog( const char * ident,
              int logopt,
              int facility );
```

Arguments:

- `ident` A string that you want to prepend to every message.
- `logopt` A bit field specifying logging options; a combination of one or more of the following values with an OR operation:
  - `LOG_CONS` If `syslog()` can’t pass the message to `syslogd`, it attempts to write the message to the `/dev/console` device. The `/dev/console` device is usually a symlink (see the `ln` command) to a real device (e.g. `/dev/text`, `/dev/con1` or `/dev/ser1`).
  - `LOG_NDELAY` Open the connection to `syslogd` immediately. Normally the opening is delayed until the first message is logged.
  - `LOG_PERROR` Write the message to standard error output as well to the system log.
  - `LOG_PID` Log the process ID with each message. This is useful for identifying instantiations of daemons.
- `facility` Encode a default facility to be assigned to all messages that don’t have an explicit facility encoded. In the following list, parameter values marked with an asterisk (*) aren’t used by any of the QNX Neutrino standard utilities.
LOG_AUTH * Authorization system.

LOG_AUTHPRIV *
Same as LOG_AUTH, but logged to a file readable only by selected individuals.

LOG_CRON * Clock daemon.

LOG_DAEMON System daemons (such as routed) that aren’t explicitly provided for by other facilities.

LOG_FTP File transfer protocol daemon.

LOG_KERN * Messages generated by the kernel. These can’t be generated by any user processes.

LOG_LPR Line printer spooling system.

LOG_MAIL Mail system.

LOG_NEWS * Network news system.

LOG_SYSLOG Messages generated internally by syslogd.

LOG_USER* Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_UUCP * The uucp system.

LOG_LOCAL0 through LOG_LOCAL7 *
Reserved for local use.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.
openlog()

Description:
The openlog() function opens the system log and provides for more specialized processing of the messages sent by syslog() and vsyslog().

Examples:
See syslog().

Classification:
POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
closelog(), setlogmask(), syslog(), vsyslog()  
logger, syslogd in the Utilities Reference
Synopsis:

#include <unix.h>

int openpty( int* amaster, 
int* aslave, 
char* name, 
struct termios* termp, 
struct winsize* winp );

Arguments:

amaster A pointer to a location where forkpty() can store the file descriptor of the master side of the pseudo-tty.

aslave A pointer to a location where forkpty() can store the file descriptor of the slave side of the pseudo-tty.

name NULL, or a pointer to a buffer where forkpty() can store the filename of the slave side of the pseudo-tty.

termp NULL, or a pointer to a termios structure that describes the terminal’s control attributes to apply to the slave side of the pseudo-tty.

winp A pointer to a winsize structure that defines the window size to use for the slave side of the pseudo-tty.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).
openpty()

Description:
The `openpty()` function finds and opens an available pseudo-tty.

Returns:
- 0  Success.
- -1 An error occurred (`errno` is set).

Errors:
- ENOENT There are no ttys available.

Classification:
Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`fork()`, `forkpty()`, `login_tty()`, `termios`
Synopsis:

```c
#include <hw/inout.h>

void out8( uintptr_t port,
           uint8_t val );
```

Arguments:

- `port` The port you want to write the value to.
- `val` The value that you want to write.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out8()` function writes an 8-bit value, specified by `val`, to the specified `port`.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
Caveats:

The calling thread must have I/O privileges; see *ThreadCtl()*’s
_STD_TCTL_IO command for details.

The calling process must also use *mmap_device_io()* to access the
device’s I/O registers.

See also:

*in8(), in8s(), in16(), in16s(), in32(), in32s(), mmap_device_io(),
out8s(), out16(), out16s(), out32(), out32s()*
**Synopsis:**

```c
#include <hw/inout.h>

void * out8s( const void * buff,
              unsigned len,
              uintptr_t port );
```

**Arguments:**

- `val` A pointer to a buffer that holds the values that you want to write.
- `len` The number of values that you want to write.
- `port` The port you want to write the values to.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `out8s()` function writes `len` 8-bit values from the buffer pointed to by `buff` to the specified `port`.

**Returns:**

A pointer to the end of the written data.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

*continued...*
Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The calling thread must have I/O privileges; see `ThreadCtl()`’s _NTO_TCTL_IO command for details.

The calling process must also use `mmap_device_io()` to access the device’s I/O registers.

See also:

`in8()`, `in8s()`, `in16()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_io()`, `out8()`, `out16()`, `out16s()`, `out32()`, `out32s()`
out16(), outbe16(), outle16()
Write a 16-bit value to a port

Synopsis:

```c
#include <hw/inout.h>

void out16( uintptr_t port,
             uint16_t val );

#define outbe16( port,
              val ) ...

#define outle16( port,
              val ) ...
```

Arguments:

- `port`  The port you want to write the value to.
- `val`   The value that you want to write.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out16()` function writes the native-endian 16-bit value, specified by `val`, to the specified `port`.

The `outbe16()` and `outle16()` macros write the native-endian 16-bit value, specified by `val`, to the specified `port` in big-endian or little-endian format, respectively.

Classification:

QNX Neutrino
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The calling thread must have I/O privileges; see `ThreadCtl()`’s
_NTO_TCTL_IO command for details.

The calling process must also use `mmap_device_io()` to access the
device’s I/O registers.

`outbe16()` and `outle16()` are implemented as macros.

See also:

`in8()`, `in8s()`, `in16()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_io()`,
`out8()`, `out8s()`, `out16s()`, `out32()`, `out32s()`
Synopsis:

```c
#include <hw/inout.h>

void * out16s( const void * buff,
               unsigned len,
               uintptr_t port );
```

Arguments:

- `val` A pointer to a buffer that holds the values that you want to write.
- `len` The number of values that you want to write.
- `port` The port you want to write the values to.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out16s()` function writes `len` words from the buffer pointed to by `buff` to the specified `port`.

Returns:

A pointer to the end of the written data.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

continued…
out16s()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The calling thread must have I/O privileges; see ThreadCtl()'s _NTO_TCTL_IO command for details.

The calling process must also use mmap_device_io() to access the device’s I/O registers.

See also:

in8(), in8s(), in16(), in16s(), in32(), in32s(), mmap_device_io(),
out8(), out8s(), out16(), out32(), out32s()
Synopsis:

```c
#include <hw/inout.h>

void out32( uintptr_t port,
             uint32_t val );

#define outbe16( port,
              val ) ...

#define outle32( port,
              val ) ...
```

Arguments:

- `port`  The port you want to write the value to.
- `val`    The value that you want to write.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out32()` function writes the 32-bit value, specified by `val`, to the specified `port`.

The `outbe32()` and `outle32()` functions macros the native-endian 32-bit value, specified by `val`, to the specified `port` in big-endian or little-endian format, respectively.

Classification:

QNX Neutrino
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The calling thread must have I/O privileges; see `ThreadCtl()`’s
_NTO_TCTL_IO command for details.

The calling process must also use `mmap_device_io()` to access the
device’s I/O registers.

$outbe16()$ and $outle16()$ are implemented as macros.

See also:

$in8()$, $in8s()$, $in16()$, $in16s()$, $in32()$, $in32s()$, `mmap_device_io()`,
$out8()$, $out8s()$, $out16()$, $out16s()$, $out32s()$
Synopsis:

```c
#include <hw/inout.h>

void * out32s( const void * buff,
               unsigned len,
               uintptr_t port );
```

Arguments:

- **val** A pointer to a buffer that holds the values that you want to write.
- **len** The number of values that you want to write.
- **port** The port you want to write the values to.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `out32s()` function writes `len` longs from the buffer pointed to by `buff` to the specified `port`.

Returns:

A pointer to the end of the written data.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

continued…
**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The calling thread must have I/O privileges; see `ThreadCtl()`’s `_NTO_TCTL_IO` command for details.

The calling process must also use `mmap_device_io()` to access the device’s I/O registers.

**See also:**

`in8()`, `in8s()`, `in16()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_io()`, `out8()`, `out8s()`, `out16()`, `out16s()`, `out32()`
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td>abort() to expm1f()</td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td>fabs() to hypotf()</td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to ltrunc()</td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td>main() to outlet32()</td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td>pathconf() to ruserok()</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td>sbrk() to system()</td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td>tan() to ynf()</td>
</tr>
</tbody>
</table>
Return the value of a configurable limit

Synopsis:

```c
#include <unistd.h>

long pathconf( const char* path, 
    int name );
```

Arguments:

- `path` The name of the file whose limit you want to get.
- `name` The name of the configurable limit; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pathconf()` function returns a value of a configurable limit indicated by `name`, which is associated with the filename given in `path`.

Configurable limits are defined in `<confname.h>`, and include at least the following values:

- `_PC_LINK_MAX`  
  Maximum value of a file’s link count.
- `_PC_MAX_CANON`  
  Maximum number of bytes in a terminal’s canonical input buffer (edit buffer).
- `_PC_MAX_INPUT`  
  Maximum number of bytes in a terminal’s raw input buffer.
- `_PC_NAME_MAX`  
  Maximum number of bytes in a file name (not including the terminating null).
Returns:

The requested configurable limit, or -1 if an error occurs (errno is set).

Errors:

EACCES Search permission is denied for a component of path.

EINVAL The name argument is invalid, or the indicated limit isn’t supported.

ELOOP Too many levels of symbolic links or prefixes.

ENAMETOOLONG The path argument, or a component of path, is too long.
pathconf()

ENOTDIR A component of the path prefix isn’t a directory.

Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    long value;

    value = pathconf( "/dev/con1", _PC_MAX_INPUT );
    printf( "Input buffer size is %ld bytes\n",
            value );
    return EXIT_SUCCESS;
}
```

Classification:
POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chown(), confstr(), errno, fpathconf(), sysconf(), termios
getconf in the Utilities Reference
Understanding System Limits chapter of the Neutrino User’s Guide
pathfind(), pathfind_r()

Search for a file in a list of directories

Synopsis:

```c
#include <libgen.h>

cchar *pathfind( const char *path,
                const char *name,
                const char *mode );

cchar *pathfind_r( const char *path,
                const char *name,
                const char *mode,
                char *buff,
                size_t buff_size );
```

Arguments:

- **path** A string that specifies the list of the directories that you want to search. The directories named in *path* are separated by colons.

- **name** The name of the file you’re looking for. If *name* begins with a slash, the name is treated as an absolute pathname, and *path* is ignored.

- **mode** A string of option letters chosen from:
  - r Readable.
  - w Writable.
  - x Executable.
  - f Normal file.
  - b Block special.
  - c Character special.
  - d Directory.
  - p FIFO (pipe).
  - u Set user ID bit.
  - g Set group ID bit.
  - k Sticky bit.
**pathfind(), pathfind_r()**

- **s**  Size nonzero.
- **buff** (pathfind_r() only) A pointer to a buffer where pathfind_r() can store the path of the file found.
- **buff_size** (pathfind_r() only) The size of the buffer that buff points to.

**Library:**

**libc**

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `pathfind()` function searches the directories named in `path` for the file `name`. The `pathfind_r()` function is a thread-safe version of `pathfind()`.

Options read, write, and execute are checked relative to the real (not the effective) user ID and group ID of the current process.

If the file `name`, with all the characteristics specified by `mode`, is found in any of the directories specified by `path`, then these functions return a pointer to a string containing the member of `path`, followed by a slash character (`/`), followed by `name`.

An empty path member is treated as the current directory. If `name` is found in the current directory, a slash isn’t prepended to it; the unadorned name is returned.

The `pathfind_r()` also includes a buffer, `buff`, and its size, `buff_size`. This buffer is used to hold the path of the file found.

**Returns:**

The path found, or NULL if the file couldn’t be found.
Examples:

Find the `ls` command using the `PATH` environment variable:

```c
pathfind (getenv ("PATH"), "ls", "rx");
```

Classification:

Unix

**pathfind()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**pathfind_r()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The string pointed to by the returned pointer is stored in an area that’s reused on subsequent calls to `pathfind()`. Don’t free this string.

Use `pathfind_r()` in multithreaded applications.
See also:

access(), getenv(), mknod(), stat()

sh in the Utilities Reference
**Synopsis:**

```c
#include <sys/pathmgr.h>

int pathmgr_symlink( const char * symlink,  
                     const char * path );
```

**Arguments:**

- `symlink`: The name of the link that you want to create.
- `path`: The path that you want to link to.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pathmgr_symlink()` function creates a symbolic link, `path`, in the process manager that redirects to the path specified by `symlink`.

The `pathmgr_unlink()` function removes the link.

---

The symbolic link isn’t permanent and is lost when the system reboots.

**Returns:**

- `0`: Success.
- `-1`: An error occurred (`errno` is set).

**Examples:**

```c
#include <stdio.h>
#include <sys/pathmgr.h>

int main(int argc, char **argv) {
```
/* Create a link /mytmp --> /dev/shmem */
if(pathmgr_symlink("/dev/shmem", "/mytmp") == -1) {
    perror("Can't make link");
}

getchar();
if(pathmgr_unlink("/mytmp") == -1) {
    perror("Can't unlink");
}

return 0;

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pathmgr_unlink(), symlink(), unlink()
Synopsis:
```
#include <sys/pathmgr.h>

int pathmgr_unlink( const char * path );
```

Arguments:

`path`  The link that you want to remove.

Library:

libc
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pathmgr_unlink()` function removes the link created by `pathmgr_symlink()`.

Returns:

0  Success.

-1  An error occurred.

Examples:

See `pathmgr_symlink()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
</table>
| Cancellation point | Yes

continued…
### pathmgr_unlink()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pathmgr_symlink()`, `symlink()`, `unlink()`
pause()

Suspend the calling thread until delivery of a signal

Synopsis:

```c
#include <unistd.h>

int pause( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pause()` function suspends the calling thread until delivery of a signal whose action is either to execute a signal handler or to terminate the process.

If the action is to terminate the process, `pause()` doesn’t return. If the action is to execute a signal handler, `pause()` returns after the signal handler returns.

Returns:

On error, `pause()` returns -1 and sets `errno` to EINTR; otherwise, it never returns.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    /* set an alarm to go off in 5 seconds */
    alarm( 5 );

    /*
    * Wait until we receive a SIGALRM signal. However,
    * since we don’t have a signal handler, any signal
    */
```
* will kill us.
*/
printf( "Hang around, "
    "waiting to die in 5 seconds\n" );
pause();
return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

alarm(), errno, sigaction()
Synopsis:

```
#include <sys/pccard.h>

int pccard_arm( pccard_t handle,
                int devtype,
                unsigned event,
                int coid );
```

Arguments:

- **handle** The handle returned by `pccard_attach()`.
- **devtype** The type of device that your application wants to be informed about. Valid devices are:
  - `PCCARD_DEV_AIMS` — Auto Incrementing Mass Storage.
  - `PCCARD_DEV_ALL` — all devices.
  - `PCCARD_DEV_FIXED_DISK` — any hard drive.
  - `PCCARD_DEV_GPIB` — General Purpose Interface Bus card.
  - `PCCARD_DEV_MEMORY` — memory type device.
  - `PCCARD_DEV_NETWORK` — any network adapter.
  - `PCCARD_DEV_PARALLEL` — PC parallel device.
  - `PCCARD_DEV_SCSI` — any SCSI interface.
  - `PCCARD_DEV_SERIAL` — 16450 serial device.
  - `PCCARD_DEV_SOUND` — any sound adapter.
  - `PCCARD_DEV_VIDEO` — any video adapter.
- **event** The type of event that you want to be notified of:
  - `PCCARD_ARM_INSERT_REMOVAL` — card insertion/removal.
- **coid** A connection ID, obtained from `ConnectAttach()`, that’s used to send the pulse.
Library:

libpccard

Use the -l pccard option to qcc to link against this library.

Description:

The `pccard_arm()` function call is used to request that the `devp-pccard` server notify the user application, via a pulse, when the specified event occurs.

Returns:

0 Success.
-1 An error occurred (errno is set).

Errors:

EBADF Invalid handle parameter.

Examples:

```c
/*
 * Ask to be informed when a Network card is inserted
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/neutrino.h>
#include <sys/pccard.h>

int main (void)
{
  pcard_t handle;
  int chid, coid;
  char buf [10];
  struct _pccard_info io;

  if ((handle = pccard_attach (0)) == -1) {
    printf ("Unable to attach to PCCARD\n");
    exit (EXIT_FAILURE);
  }

  return 0;
}
```
if ((chid = ChannelCreate (_NTO_CHF_FIXED_PRIORITY)) == -1) {
    printf ("Unable to create channel\n");
    exit (EXIT_FAILURE);
}

if ((coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL,
            0)) == -1) {
    printf ("Unable to ConnectAttach\n");
    exit (EXIT_FAILURE);
}

if (pccard_arm (handle, _PCCARD_DEV_NETWORK,
            _PCCARD_ARM_INSERT_REMOVE, coid) == -1) {
    perror ("Arm failed");
    exit (EXIT_FAILURE);
}

/* To be informed about any card insertion/removal event,
 * change _PCCARD_DEV_NETWORK to _PCCARD_DEV_ALL.
 */

/* */
/* */
/* */
/* */
/* Get information from socket 0 - function 0 */
if (pccard_info (handle, 0, &io, sizeof (io)) == -1) {
    perror ("Info failed");
    exit (EXIT_FAILURE);
}

if (io.flags & _PCCARD_FLAG_CARD) {
    printf ("Card inserted in socket 1 - Type %x\n",
            io.window [0].device & 0xff00);
    /* Now lock the card in socket 1 with exclusive access */
    if (pccard_lock (handle, 0, 0, O_RDWR | O_EXCL) == -1) {
        perror ("Lock failed");
        exit (EXIT_FAILURE);
    }
    /* Read 2 bytes of the CIS from offset 0 in attribute memory */
    if (pccard_raw_read (handle, 0, _PCCARD_MEMTYPE_ATTRIBUTE,
                    0, 2, buf) == -1) {
        perror ("Raw read");
        exit (EXIT_FAILURE);
    }
    /* More user logic... */
}

pccard_unlock (handle, 0, 0);
pccard_detach (handle);
```c
return (EXIT_SUCCESS);
}
```

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pccard_attach(), pccard_detach(), pccard_info(), pccard_lock(), pccard_raw_read(), pccard_unlock()`
Synopsis:

```
#include <sys/pccard.h>

pccard_t pccard_attach( int reserved );
```

Arguments:

`reserved` Pass 0 for this argument.

Library:

`libpccard`

Use the `-l pccard` option to `qcc` to link against this library.

Description:

The `pccard_attach()` function attaches a user application to the `devp-pccard` server. You must call this function before using any of the other PC card functions, because it returns a handle that all the other PC Card functions use.

Returns:

- `>0` A value to be used as `handle` in all other PC Card function calls.
- `-1` Can’t locate the `devp-pccard` server.
- `-2` Send to `devp-pccard` server failed.
- `-3` The `devp-pccard` server returned an error (`errno` is set).

Errors:

- `EBUSY` The `devp-pccard` server is unable to service this request.
**Examples:**

See *pccard_arm()*.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*pccard_arm(), pccard_detach(), pccard_info(), pccard_lock(), pccard_raw_read(), pccard_unlock()*
Synopsis:
#include <sys/pccard.h>

int pccard_detach( pccard_t handle );

Arguments:

handle The handle returned by pccard_attach().

Library:
libpccard
Use the -l pccard option to qcc to link against this library.

Description:
The pccard_detach() function detaches the user application from the
devp-pccard server. Any locks that you previously applied with
pccard_lock() are freed.

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EBADF Invalid handle parameter.

Examples:
See pccard_arm().

Classification:
QNX Neutrino
**See also:**

`pccard_arm()`, `pccard_attach()`, `pccard_info()`, `pccard_lock()`, `pccard_raw_read()`, `pccard_unlock()`
Obtain socket information from the devp-pccard server

Synopsis:

```c
#include <sys/pccard.h>

int pccard_info( pccard_t handle,
    int socket,
    struct _pccard_info* info,
    unsigned size );
```

Arguments:

- `handle`: The handle returned by `pccard_attach()`.
- `socket`: Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.
- `info`: A pointer to a `_pccard_info` structure that the function fills with the socket information. For more information, see below.
- `size`: Size of the `_pccard_info` structure.

Library:

```
libpccard
```

Use the `-l pccard` option to `qcc` to link against this library.

Description:

The `pccard_info()` function call retrieves socket setup information from the devp-pccard server. The information is returned in the `_pccard_info` structure.

`_pccard_info` structure

The `_pccard_info` structure is defined in `<pccard.h>` as:

```c
struct _pccard_info {
    int16_t socket;    // Socket number (0 based)
    uint16_t status;   // Card status (from socket services spec)
};
```
Returns:

A positive integer

Success. The socket parameter is returned.

-1 An error occurred (errno is set).

Errors:

ENODEV Invalid socket parameter.
Examples:

See \texttt{pccard\_arm()}.  

Classification:  

QNX Neutrino  

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:  

\texttt{pccard\_arm()}, \texttt{pccard\_attach()}, \texttt{pccard\_detach()}, \texttt{pccard\_lock()},  
\texttt{pccard\_raw\_read()}, \texttt{pccard\_unlock()}
**pccard_lock()**  
Lock the window of the card in the selected socket

**Synopsis:**

```c
#include <sys/pccard.h>

int pccard_lock( pcard_t handle,  
                 int socket,  
                 int index,  
                 int oflag );
```

**Arguments:**

- `handle`: The handle returned by `pccard_attach()`.
- `socket`: Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.
- `index`: The window/function number that you want to lock. You can get the window number from the `_pccard_info` structure (see `pccard_info()`).
- `oflag`: Created by ORing the values required (e.g. `O_RDWR | O_EXCL`) for read/write and exclusive access.

**Library:**

`libpccard`

Use the `-l pccard` option to `qcc` to link against this library.

**Description:**

The `pccard_lock()` function call provides exclusive or shared access to the PC Card in `socket` and also sets access permissions.
pccard_lock()

Returns:

A positive integer
   Success.
-1  An error occurred (errno is set).

Errors:

EBADF   Invalid handle parameter.
EBUSY   The window is already locked by another process.
ENODEV  Invalid socket parameter, no PC Card is present in the
         socket, or invalid index parameter.

Examples:

See pccard_arm().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pccard_arm(), pccard_attach(), pccard_detach(), pccard_info(),
pccard_raw_read(), pccard_unlock()
**Synopsis:**

```c
#include <sys/pccard.h>

ssize_t pccard_raw_read( pccard_t handle, int socket, int type, unsigned addr, ssize_t len, void* buf );
```

**Arguments:**

- **handle** The handle returned by `pccard_attach()`.
- **socket** Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.
- **type** The type of memory that you want to read. Valid values are:
  - `_PCCARD_MEMTYPE_COMMON`
  - `_PCCARD_MEMTYPE_ATTRIBUTE`
- **addr** The memory address that you want to read from the CIS.
- **len** The size of the memory that you want to read.
- **buf** A pointer to a buffer where the function can store the information that it reads from the PC Card.

**Library:**

`libpccard`

Use the `-l pccard` option to `qcc` to link against this library.
Description:

The `pccard_raw_read()` function returns the raw CIS (Card Information Structure) data from the PC Card.

Returns:

A positive integer

Success. The length read is returned.

-1 An error occurred (`errno` is set).

Errors:

EBADF Invalid `handle` parameter.

ENODEV Invalid `socket` parameter.

Examples:

See `pccard_arm()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`pccard_arm()`, `pccard_attach()`, `pccard_detach()`, `pccard_info()`,
`pccard_lock()`, `pccard_unlock()`
pccard_unlock()

Unlock the window of the card in the selected socket

Synopsis:

```c
#include <sys/pccard.h>

int pccard_unlock( pccard_t handle,  
                   int socket,          
                   int index );
```

Arguments:

- `handle` The handle returned by `pccard_attach()`.
- `socket` Contains both the socket number as well as the function within the socket. This is achieved by shifting the function number left 8 bits and ORing it with the socket number. The socket number is zero-based.
- `index` The window/function number that you want to unlock. You can get the window number from the `_pccard_info` structure (see `pccard_info()`).

Library:

```c
libpccard
```

Use the `-l pccard` option to `qcc` to link against this library.

Description:

The `pccard_unlock()` function unlocks a window previously locked by a call to `pccard_lock()`. It can only unlock a window locked by the same process ID — you can’t unlock a window locked by another process.

Returns:

- `0` Success.
- `-1` An error occurred (`errno` is set).
Errors:

- EBADF: Invalid handle parameter.
- ENODEV: Invalid socket parameter, no PC Card is present in the socket, or invalid index parameter.

Examples:

See `pccard_arm()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pccard_arm()`, `pccard_attach()`, `pccard_detach()`, `pccard_info()`, `pccard_lock()`, `pccard_raw_read()`
pci_attach()
Connect to the PCI server

Synopsis:

```c
#include <hw/pci.h>

int pci_attach( unsigned flags );
```

Arguments:

- `flags` There are currently no flags defined for this function.

Library:

- `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_attach()` function connects to the Peripheral Component Interconnect (PCI) server.

- You must call `pci_attach()` before calling any of the other PCI functions.

Returns:

- A handle used for calling `pci_detach()`, or `-1` if an error occurs.

Errors:

- See `open()`.

Classification:

- QNX Neutrino
pci_attach()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pci_attach_device(), pci_detach(), pci_detach_device(),
pci_find_class(), pci_find_device(), pci_present(),
pci_read_config(), pci_read_config8(), pci_read_config16(),
pci_read_config32(), pci_rescan_bus(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()

“Peripheral Component Interconnect (pci-*)” in the Utilities
Summary chapter of the Utilities Reference


**Synopsis:**

```c
#include <hw/pci.h>

void* pci_attach_device(
    void* handle,
    uint32_t flags,
    uint16_t idx,
    struct pci_dev_info* info);
```

**Arguments:**

- **handle** A handle that identifies the device. The first time you call this function, set `handle` to NULL. This function returns a handle that you can use in a subsequent call to allocate resources for the device.

- **flags** Flags that tell the PCI server how you want it to handle resources, which resources to scan for, and which resources to allocate; see “Flags,” below.

- **idx** The index of the device: 0 for the first device, 1 for the second, and so on.

- **info** A pointer to a `pci_dev_info` structure (see below) that specifies the class code, vendor/device ID, or bus number and device/function number that you want to scan for. The function fills in this structure with information about the device.

**Library:**

```c
libc
```

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `pci_attach_device()` function attaches a driver to a PCI device.

---

You must successfully call `pci_attach()` before calling any of the other PCI functions.

---

Typically drivers use this function to attach themselves to a PCI device, so that other drivers can’t attach to the same device. If you specify the PCI_SHARE flag (see “Flags,” below), then multiple drivers can attach to the same device.

The server can scan based on a class code, vendor/device ID, or bus number and device/function number. To control the server scanning, initialize the appropriate fields of the `info` structure and set the appropriate flags.

When you first attach to an uninitialized device, the PCI server assigns all the I/O ports, memory and IRQs required for the device. It also does the IRQ routing for you. Once this has completed successfully, it fills in all these values into your `pci_dev_info` structure to return these values to your application.

When a driver attaches to a device, the PCI server allocates the necessary resources for the device from `procnto` using the `rsrcdbmgr*` calls. On X86 BIOS systems, these resources are normally allocated by the BIOS, but on non-x86 systems, these resources have to be allocated from `procnto`.

You can detach the device by passing its handle to `pci_detach_device()`. If you call `pci_detach()`, any resources that `pci_attach_device()` allocates are freed.

**pci_dev_info structure**

This function fills in a `pci_dev_info` structure that describes an occurrence of a device.
The `pci_attach_device()` function doesn’t map any of the I/O or memory regions into the process’s address space. The addresses returned in the `pci_dev_info` structure are all physical addresses.

This structure has the following members:

```c
uint16_t DeviceId
    The device ID (input/output). For a list of supported device IDs, see `<hw/pci_devices.h>`.

uint16_t VendorId
    The vendor ID (input/output). For a list of supported vendor IDs, see `<hw/pci_devices.h>`.

uint16_t SubsystemId
    The subsystem ID (output).

uint16_t SubsystemVendorId
    The subsystem vendor ID (output).

uint8_t BusNumber
    The bus number (input/output).

uint8_t DevFunc
    The device/function number (input/output).

uint8_t Revision
    The device revision (output).

uint32_t Class
    The class code (input/output). For a list of class codes, see `<hw/pci.h>`. This field is an ORed combination of a class code and a subclass code (e.g. `PCI_CLASS_DISPLAY | PCI_SUBCLASS_DISPLAY_XGA`).
```
uint32_t *Irq  The interrupt number (output).

uint64_t *CpuIoTranslation
The CPU-to-PCI translation value (pci_addr = cpu_addr - translation).

uint64_t *CpuMemTranslation
The CPU-to-PCI memory translation (pci_addr = cpu_addr - translation).

uint64_t *CpuIsaTranslation
The CPU-to-ISA memory translation (pci_addr = cpu_addr - translation).

uint64_t *CpuBmstrTranslation
The translation from the CPU busmaster address to the PCI busmaster address (pci_addr = cpu_addr + translation).

uint64_t *PciBaseAddress [6]
The PCI base address (array of six uint64_t items).

This function decodes bits 1 and 2 to see whether the register is 32 or 64 bits wide, hence the 64-bit values for the base registers.

uint64_t *CpuBaseAddress [6]
The CPU base address (an array of six uint64_t items).

Some platforms translate addresses across PCI bridges, so that there’s one address on the PCI side of the bridge and another on the CPU side. Under x86, the PciBaseAddress and CpuBaseAddress are the same, but under other platforms, these will be different. In your user application you should always use the CpuBaseAddress.
void pci_attach_device()
{
  uint32_t BaseAddressSize [6]
  The size of the base address aperture into the board (an array of six uint32_t items).

  uint64_t PciRom
  The PCI ROM address.

  uint64_t CpuRom
  The CPU ROM address.

  uint32_t RomSize
  The size of the aperture into the board.

Flags

The flags parameter tells the PCI server how resources are to be handled, which resources to scan for, and which resources to allocate.

These bits control how resources are handled:

PCI_SHARE Allow resources to be shared with other drivers. If this isn’t set, no other driver can attach to the device.

PCI_PERSIST Resources persist after the device is detached.

The following bits ask the PCI server to scan for a device based on the fields that you specified in the structure pointed to by info:

PCI_SEARCH_VEND
  VendorId

PCI_SEARCH_VENDEV
  DeviceId and VendorId

PCI_SEARCH_CLASS
  Class
PCI_ATTACH_DEVICE()

PCI_SEARCH_BUSDEV
   BusNumber and DevFunc

These bits specify which members of the structure the server should initialize:

PCI_INIT_IRQ    Iirq
PCI_INIT_ROM    PciRom and CpuRom
PCI_INIT_BASE0 ... PCI_INIT_BASE5
   The specified entries of the PciBaseAddress and CpuBaseAddress arrays
PCI_INIT_ALL    All members except PciRom and CpuRom

The bits also include:

PCI_MASTER_ENABLE
   Enable bus mastering on the device.

If you pass 0 for the flags, the default is PCI_SEARCH_VENDEV.

Testing and converting addresses

To facilitate the testing of addresses returned by the PCI server, at least the following macros are defined in the <hw/pci.h> header file:

PCI_IS_IO( address )
   Test whether the address is an I/O address.

PCI_IS_MEM( address )
   Test whether the address is a memory address.

PCI_IOADDR( address )
   Convert the address returned by the PCI server to an I/O address.
pci_attach_device()

**PCI_MEM_ADDR(address)**

Convert the address returned by the PCI server to a memory address.

**PCI_ROM_ADDR(address)**

Convert the address returned by the PCI server to a ROM address.

For example:

```c
{
    uint64_t   port;

    /* Test the address returned by the pci server */
    if (PCI_IS_IO(addr))
        port = (PCI_IO_ADDR(addr));
}
```

**Returns:**

A handle to be used for other pci_* calls associated with a handle, or NULL if an error occurs (errno is set).

**Errors:**

- **EBUSY** An application has already attached to the device. If it’s safe to share the device, specify PCI_SHARE in the flags field.
- **EINVAL** The function couldn’t attach a resource to the device.
- **ENODEV** This device wasn’t found.

**Examples:**

Attach to and allocate all resources for the first occurrence of an Adaptec 2940 adapter:

```c
#include <hw/pci.h>
#include <hw/pci_devices.h>
#include <stdio.h>
```
#include <stdlib.h>

int main( void )
{
    int pidx;
    void* hdl;
    int phdl;
    struct pci_dev_info inf;

    /* Connect to the PCI server */
    phdl = pci_attach( 0 );
    if( phdl == -1 ) {
        fprintf( stderr, "Unable to initialize PCI\n" );
    return EXIT_FAILURE;
    }

    /* Initialize the pci_dev_info structure */
    memset( &inf, 0, sizeof( inf ) );
    pidx = 0;
    inf.VendorId = PCI_VENDOR_ID_ADAPTEC;
    inf.DeviceId = PCI_DEVICE_ID_ADAPTEC_2940F;

    hdl = pci_attach_device( NULL, PCI_INIT_ALL, pidx, &inf );
    if( hdl == NULL ) {
        fprintf( stderr, "Unable to locate adapter\n" );
    } else {
        /* Do something to the adapter */
        pci_detach_device( hdl );
    }

    /* Disconnect from the PCI server */
    pci_detach( phdl );

    return EXIT_SUCCESS;
}

Attach to the first occurrence of an Adapter 2940 adapter and allocate resources in a second call:

#include <hw/pci.h>
#include <hw/pci_devices.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int pidx;
    void* hdl;
    void* retval;

```c
int phdl;
struct pci_dev_info inf;

phdl = pci_attach( 0 );
if( phdl == -1 ) {
    fprintf( stderr, "Unable to initialize PCI\n" );

    return EXIT_FAILURE;
}

memset( &inf, 0, sizeof( inf ) );
pidx = 0;
inf.VendorId = PCI_VENDOR_ID_ADAPTEC;
inf.DeviceId = PCI_DEVICE_ID_ADAPTEC_2940F;

hdl = pci_attach_device( NULL, 0, pidx, &inf );
if( hdl == NULL ) {
    fprintf( stderr, "Unable to locate adapter\n" );
}

retval = pci_attach_device( hdl, PCI_INIT_ALL, pidx, &inf );
if( retval == NULL ) {
    fprintf( stderr, "Unable allocate resources\n" );
}

pci_detach( phdl );

return EXIT_SUCCESS;
```

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

pci_attach(), pci_detach(), pci_detach_device(), pci_find_class(),
pci_find_device(), pci_present(), pci_read_config(),
pci_read_config8(), pci_read_config16(), pci_read_config32(),
pci_rescan_bus(), pci_write_config()
**pci_detach()**
Disconnect from the PCI server

**Synopsis:**

```c
#include <hw/pci.h>

int pci_detach( unsigned handle );
```

**Arguments:**

*handle*  The value returned by a successful call to *pci_attach()*.

**Library:**

*libc*

Use the `-l c` option to *qcc* to link against this library. This library is usually included automatically.

**Description:**

The *pci_detach()* function disconnects from the PCI server. Any resources allocated with *pci_attach_device()* are released.

The *pci_attach()* function opens a file descriptor against the PCI server, and all of the low-level library calls to the PCI server use this fd. When you call *pci_detach()* , the low-level code does a *close()* on the file descriptor, which tells the PCI server to clean up any allocations associated with it.

---

Don’t call any of the other *pci_*() functions after calling *pci_detach()* (unless you’ve reattached with *pci_attach()*).

**Returns:**

*PCI_SUCCESS*.

**Classification:**

QNX Neutrino
pci_detach()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pci_attach(), pci_attach_device(), pci_detach_device(),
pci_find_class(), pci_find_device(), pci_present(),
pci_read_config(), pci_read_config8(), pci_read_config16(),
pci_read_config32(), pci_rescan_bus(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
**pci_detach_device()**

*Detach a driver from a PCI device*

**Synopsis:**

```c
#include <hw/pci.h>

int pci_detach_device( void* handle );
```

**Arguments:**

- **handle**
  The handle returned by `pci_attach_device()`.

**Library:**

- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pci_detach_device()` function detaches a driver from a PCI device. Any resources allocated with `pci_attach_device()` are released, unless you attached the device with the PCI_PERSIST flag set.

You must successfully call `pci_attach()` before calling any of the other PCI functions.

**Returns:**

- **PCI_DEVICE_NOT_FOUND**
  No device could be found for `handle`.

- **PCI_SUCCESS**
  Success.

- `-1` You haven’t called `pci_attach()`, or the call to it failed.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pci_attach(), pci_attach_device(), pci_detach(), pci_find_class(),
pci_find_device(), pci_present(), pci_read_config(),
pci_read_config8(), pci_read_config16(), pci_read_config32(),
pci_rescan_bus(), pci_write_config(), pci_write_config8(),
pci_write_config16(), pci_write_config32()
**Synopsis:**

```c
#include <hw/pci.h>

int pci_find_class( unsigned long class_code,  
                    unsigned index,  
                    unsigned* bus,   
                    unsigned* dev_func );
```

**Arguments:**

- **class_code**  The class of device or function that you want to find. For a list of class codes, see `<hw/pci.h>`. You must OR together a class code and a subclass code (e.g. `PCI_CLASS_DISPLAY | PCI_SUBCLASS_DISPLAY_XGA`).
- **index**  The index of the device or function that you want to find: 0 for the first, 1 for the second, and so on.
- **bus**  The bus number, in the range [0...255].
- **dev_func**  The device or function number of the nth device or function of the given class. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pci_find_class()` function determines the location of the nth PCI device or function that has the specified class code.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

You can find all the devices having the same class code by making successive calls to this function, starting with an index of 0, and incrementing it until `PCI_DEVICE_NOT_FOUND` is returned.

**Returns:**

- `PCI_DEVICE_NOT_FOUND`
  - The device or function wasn’t found.
- `PCI_SUCCESS`
  - The device or function was found.
- -1 You haven’t called `pci_attach()`, or the call to it failed.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pci_attach()`, `pci_attach_device()`, `pci_detach()`,
`pci_detach_device()`, `pci_find_device()`, `pci_present()`,
`pci_read_config()`, `pci_read_config8()`, `pci_read_config16()`,
pci_find_class()

pci_read_config32(), pci_rescan_bus(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
Synopsis:

```c
#include <hw/pci.h>

int pci_find_device( unsigned device,
                     unsigned vendor,
                     unsigned index,
                     unsigned* bus,
                     unsigned* dev_func );
```

Arguments:

- `device`: The device ID. For a list of supported device IDs, see `<hw/pci_devices.h>`.
- `vendor`: The vendor ID. For a list of supported vendor IDs, see `<hw/pci_devices.h>`.
- `index`: The index (n) of the device or function sought.
- `bus`: A pointer to a location where the function can store the bus number of the device or function found.
- `dev_func`: A pointer to a location where the function can store the device or function ID of the nth device or function found with the specified device and vendor IDs. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_find_device()` function returns the location of the nth PCI device that has the specified device and vendor IDs.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

You can find all the devices having the same device and vendor IDs by making successive calls to this function, starting with an index of 0, and incrementing it until `PCI_DEVICE_NOT_FOUND` is returned.

**Returns:**

- **PCI_DEVICE_NOT_FOUND**
  The device or function wasn’t found.

- **PCI_SUCCESS**
  The device or function was found.

- **-1**
  You haven’t called `pci_attach()`, or the call to it failed.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pci_attach()`, `pci_attach_device()`, `pci_detach()`,
`pci_detach_device()`, `pci_find_class()`, `pci_present()`,
`pci_read_config()`, `pci_read_config8()`, `pci_read_config16()`,
`pci_find_device()`

`pci_read_config32()`, `pci_rescan_bus()`, `pci_write_config()`,
`pci_write_config8()`, `pci_write_config16()`, `pci_write_config32()`
pci_irq_routing_options() © 2007, QNX Software Systems GmbH & Co. KG.

Retrieve PCI IRQ routing information

Synopsis:

```c
#include <hw/pci.h>

int pci_irq_routing_options(
    IRQRoutingOptionsBuffer * buf,
    uint32_t * irq);
```

Arguments:

- `buf` A pointer to an `IRQRoutingOptionsBuffer` structure where the function can store the IRQ routing information. For information about the layout of this buffer, see PCI BIOS SPECIFICATION Revision 2.1. You can get it from the PCI Special Interest Group at [http://pcisig.com/](http://pcisig.com/).

- `irq` A pointer to a location where the function can store the current state of interrupts.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_irq_routing_options()` function returns the following:

- PCI interrupt routing options available on the system motherboard
- the current state of interrupts that are currently exclusively assigned to PCI.

Routing information is returned in a data buffer that contains an IRQ routing for each PCI device or slot.
You must successfully call `pci_attach()` before calling any of the other PCI functions. The `pci_irq_routing_options()` function is for x86 only.

**Returns:**

- `PCI_SUCCESS` Success.
- `-1` You haven’t called `pci_attach()`, or the call to it failed.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <inttypes.h>
#include <hw/pci.h>
#include <sys/neutrino.h>

struct {
    IRQRoutingOptionsBuffer buf;
    uint8_t databuf [2048];
} route_buf;

int main (void)
{
    int phdl;
    uint32_t irq;

    if ((phdl = pci_attach (0)) == -1) {
        printf (“Unable to attach - errno %s\n", strerror (errno));
        exit (1);
    }

    memset (route_buf.databuf, 0, sizeof (route_buf.databuf));
    route_buf.buf.BufferSize = sizeof (route_buf.databuf);
    if (pci_irq_routing_options (&route_buf.buf, &irq) !=
        PCI_SUCCESS) {
        printf (“Routing option failed - errno %s\n", strerror (errno));
        exit (1);
    }

    printf (“PCI Iq Map = %x\n”, irq);
```
\textit{pci_irq_routing_options()} \hspace{1em} © 2007, QNX Software Systems GmbH & Co. KG.

\begin{verbatim}
pci_detach (phdl);
return (0);
}
\end{verbatim}

\section*{Classification:}

QNX Neutrino

\begin{center}
\begin{tabular}{ll}
\textbf{Safety} & \\
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}
\end{center}

\section*{See also:}

\begin{verbatim}
pci_attach(), pci_attach_device(), pci_detach(),
pci_detach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config(), pci_read_config8(),
pci_read_config16(), pci_read_config32(), pci_rescan_bus(),
pci_write_config(), pci_write_config8(), pci_write_config16(),
pci_write_config32()
\end{verbatim}
pci_map_irq()
Map an interrupt pin to an IRQ

Synopsis:
#include <hw/pci.h>

int pci_map_irq( unsigned bus,
                 unsigned dev_func,
                 short intno,
                 short intpin );

Arguments:
bus The bus number of the device.
dev_func The device or function number of the device. The
device number is in bits 7 through 3, and the function
number is in bits 2 through 0.
intno The interrupt to be mapped (e.g. 0 - 15 on x86).
intpin The PCI interrupt pin (0x0a - 0x0d).

Library:
libc
Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:
The pci_map_irq() function maps a PCI interrupt pin to a specific
interrupt request (IRQ).

You must successfully call pci_attach() before calling any of the
other PCI functions.

Returns:
PCI_SUCCESS
Success.
PCI_SET_FAILED
   The PCI server was unable to map the intnolintpin.
PCI_UNSUPPORTED_FUNCTION
   This function isn’t supported.
-1   You haven’t called pci_attach(), or the call to it failed.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pci_attach(), pci_attach_device(), pci_detach(),
pci_detach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config(), pci_read_config8(),
pci_read_config16(), pci_read_config32(), pci_rescan_bus(),
pci_write_config(), pci_write_config8(), pci_write_config16(),
pci_write_config32()
Synopsis:

```c
#include <hw/pci.h>

int pci_present( unsigned* lastbus,
                 unsigned* version,
                 unsigned* hardware );
```

Arguments:

- `lastbus` The number of the last PCI bus in the system. PCI buses are numbered from 0, up to and including this value.
- `version` The version number of the PCI interface.
- `hardware` The specific hardware characteristics the platform supports with regard to accessing configuration space and generating PCI Special Cycles.

The PCI specification defines two hardware mechanisms for accessing configuration space. Bit 0 of `hardware` is set (1) if mechanism 1 is supported, and reset (0) otherwise. Bit 1 is set (1) if mechanism 2 is supported, and reset (0) otherwise.

The specification also defines hardware mechanisms for generating Special Cycles. Bit 4 of `hardware` is set (1) if the platform supports Special Cycle generation based on Config Mechanism 1, and reset (0) otherwise. Bit 5 is set (1) if the platform supports Special Cycle generation based on Config Mechanism 2, and reset (0) otherwise.

The arguments can be NULL if you just want to check for PCI capabilities.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `pci_present()` function determines whether or not the PCI BIOS interface function set is present. It also determines the following:

- the current interface version

- what hardware mechanism for accessing configuration space is supported

- whether or not the hardware supports the generation of PCI Special Cycles.

You must successfully call `pci_attach()` before calling any of the other PCI functions.

Returns:

-1   PCI BIOS isn’t present.

`PCI_SUCCESS`

    PCI BIOS is present.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

*pci_attach*, *pci_attach_device*, *pci_detach*,
*pci_detach_device*, *pci_find_class*, *pci_find_device*,
*pci_read_config*, *pci_read_config8*, *pci_read_config16*,
*pci_read_config32*, *pci_rescan_bus*, *pci_write_config*,
*pci_write_config8*, *pci_write_config16*, *pci_write_config32*
**Synopsis:**

```c
#include <hw/pni.h>

int pci_read_config( void* handle,
                     unsigned offset,
                     unsigned count,
                     size_t size,
                     void* buff );
```

**Arguments:**

- `handle`: The handle returned by `pci_attach_device()`.
- `offset`: The offset into the configuration space where you want to read from.
- `count`: The number of objects that you want to read.
- `size`: The size of each object.
- `buff`: A pointer to a buffer where the function can store the objects that it reads.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pci_read_config()` function reads `count` objects of the specified `size` into `buff` at the given `offset` from the configuration space of the PCI device specified by `handle`. 
You must successfully call `pci_attach()` before calling any of the other PCI functions.

Returns:

- **PCI_BAD_REGISTER_NUMBER**
  The offset is invalid.

- **PCI_BUFFER_TOO_SMALL**
  The PCI BIOS server reads only 100 bytes at a time; `size` is too large.

- **PCI_DEVICE_NOT_FOUND**
  The `handle` is invalid.

- **PCI_SUCCESS**
  Success.

-1 You haven’t called `pci_attach()`, or the call to it failed.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

pci_attach(), pci_attach_device(), pci_detach(),
pci_detach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config8(), pci_read_config16(),
pci_read_config32(), pci_rescan_bus(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
Synopsis:
#include <hw/pci.h>

int pci_read_config8( unsigned bus,
unsigned dev_func,
unsigned offset,
unsigned count,
char* buff );

Arguments:

bus The bus number.

dev_func The name of the device or function.

offset The register offset into the configuration space, in the
range [0...255].

count The number of bytes to read.

buff A pointer to a buffer where the requested bytes are
placed.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The pci_read_config8() function reads the specified number of bytes
from the configuration space of the given device or function.

💡 You must successfully call pci_attach() before calling any of the
other PCI functions.
Returns:

PCI_BAD_REGISTER_NUMBER
An invalid register offset was given.

PCI_BUFFER_TOO_SMALL
The PCI BIOS server reads only 100 bytes at a time; count is too large.

PCI_SUCCESS
The device or function was found.

-1 You haven’t called pci_attach(), or the call to it failed.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pci_attach(), pci_attach_device(), pci_detach(),
pci_detach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config(), pci_read_config16(),
pci_read_config32(), pci_rescan_bus(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
pci_read_config16()

Read 16-bit values from the configuration space of a device

Synopsis:

```c
#include <hw/pci.h>

int pci_read_config16( unsigned bus,
                        unsigned dev_func,
                        unsigned offset,
                        unsigned count,
                        char* buff );
```

Arguments:

- `bus` The bus number.
- `dev_func` The name of the device or function.
- `offset` The register offset into the configuration space. This offset must be aligned to a 16-bit boundary (that is 0, 2, 4, ..., 254 bytes).
- `count` The number of 16-bit values to read.
- `buff` A pointer to a buffer where the requested 16-bit values are placed.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_read_config16()` function reads the specified number of 16-bit values from the configuration space of the given device or function.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

**Returns:**

- `PCI_BAD_REGISTER_NUMBER`
  - An invalid offset register number was given.

- `PCI_BUFFER_TOO_SMALL`
  - The PCI BIOS server reads only 50 words at a time; `count` is too large.

- `PCI_SUCCESS`
  - The device or function was found.

- `-1`
  - You haven’t called `pci_attach()`, or the call to it failed.

**Classification:**

- **QNX Neutrino**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
pci_read_config16()

See also:

pci_attach(), pci_attach_device(), pci_detach(),
pci_detach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config(), pci_read_config8(),
pci_read_config32(), pci_rescan_bus(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
pci_read_config32() © 2007, QNX Software Systems GmbH & Co. KG.

Read 32-bit values from the configuration space of a device

Synopsis:

```c
#include <hw/pci.h>

int pci_read_config32( unsigned bus,
                       unsigned dev_func,
                       unsigned offset,
                       unsigned count,
                       char* buff );
```

Arguments:

- `bus` The bus number.
- `dev_func` The name of the device or function.
- `offset` The register offset into the configuration space. This offset must be aligned to a 32-bit boundary (that is 0, 4, 8, ..., 252 bytes).
- `count` The number of 32-bit values to read.
- `buff` A pointer to a buffer where the requested 32-bit values are placed.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_read_config32()` function reads the specified number of 32-bit values from the configuration space of the given device or function.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

**Returns:**

- **PCI_BAD_REGISTER_NUMBER**
  - An invalid register offset was given.
- **PCI_SUCCESS**
  - The device or function was found.
- **-1**
  - You haven’t called `pci_attach()`, or the call to it failed.

**Classification:**

- QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `pci_attach()`, `pci_attach_device()`, `pci_detach()`,
- `pci_detach_device()`, `pci_find_class()`, `pci_find_device()`,
- `pci_present()`, `pci_read_config()`, `pci_read_config8()`,
- `pci_read_config16()`, `pci_rescan_bus()`, `pci_write_config()`,
- `pci_write_config8()`, `pci_write_config16()`, `pci_write_config32()`
Synopsis:

```c
#include <hw/pci.h>

int pci_rescan_bus( void );
```

Library:

libc

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_rescan_bus()` function asks the PCI server to rescan the PCI bus(es) for devices that have been inserted or removed. This is used in hot swap situations such as for CardBus cards. The PCI server updates its internal configuration to reflect any changes.

You must successfully call `pci_attach()` before calling any of the other PCI functions.

Returns:

- `PCI_SUCCESS` Success.
- `-1` The function failed.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
pci_rescan_bus()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pci_attach(), pci_attach_device(), pci_detach(),
pciDetach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config(), pci_read_config8(),
pci_read_config16(), pci_read_config32(), pci_write_config(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
**pci_write_config()**

Write to the configuration space of a PCI device

**Synopsis:**
```
#include <hw/pci.h>

int pci_write_config( void* handle, 
                       unsigned offset, 
                       unsigned count, 
                       size_t size, 
                       const void* buff );
```

**Arguments:**
- **handle** The handle returned by `pci_attach_device()`.
- **offset** The offset into the configuration space where you want to write the data.
- **count** The number of objects that you want to write.
- **size** The size of each object.
- **buff** A pointer to the data that you want to write.

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pci_write_config()` function writes `count` objects of the specified `size` from `buff` at the given `offset` to the configuration space of the PCI device specified by `handle`.

---

You must successfully call `pci_attach()` before calling any of the other PCI functions.
PCI write_config()

Returns:

PCI_BAD_REGISTER_NUMBER
   The offset specified is invalid.
PCI_BUFFER_TOO_SMALL
   The size argument is too large.

PCI_SET FAILED
   An error occurred writing to the configuration space of the device.

PCI_SUCCESS
   Success.

PCI_UNSUPPORTED FUNCT
   This device doesn’t support writing to its configuration space.

-1 You haven’t called pci_attach(), or the call to it failed.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

pci_attach(), pci_attach_device(), pci_detach(),
pci_detach_device(), pci_find_class(), pci_find_device(),
pci_present(), pci_read_config(), pci_read_config8(),
pci_write_config()  © 2007, QNX Software Systems GmbH & Co. KG.

pci_read_config16(), pci_read_config32(), pci_rescan_bus(),
pci_write_config8(), pci_write_config16(), pci_write_config32()
pci_write_config8()
Write bytes to the configuration space of a PCI device

Synopsis:
#include <hw/pci.h>

int pci_write_config8( unsigned bus,
                        unsigned dev_func,
                        unsigned offset,
                        unsigned count,
                        char* buff );

Arguments:

bus The bus number.
dev_func The device or function ID. The device number is in bits 7 through 3, and the function number in bits 2 through 0.
offset The register offset into the configuration space, in the range [0...255].
count The number of bytes to write.
buff A pointer to a buffer containing the data to be written into the configuration space.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The pci_write_config8() function writes individual bytes to the configuration space of the specified device.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

Returns:

- **PCI_BAD_REGISTER_NUMBER**
  - An invalid offset register number was given.
- **PCI_BUFFER_TOO_SMALL**
  - The `size` argument is greater than 100 bytes.
- **PCI_SUCCESS**
  - The device or function was found.
- **-1**
  - You haven’t called `pci_attach()`, or the call to it failed.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `pci_attach()`, `pci_attach_device()`, `pci_detach()`, `pci_detach_device()`, `pci_find_class()`, `pci_find_device()`, `pci_present()`, `pci_read_config()`, `pci_read_config8()`,
pci_write_config8()
**Synopsis:**

```c
#include <hw/pci.h>

int pci_write_config16( unsigned bus,
    unsigned dev_func,
    unsigned offset,
    unsigned count,
    char* buff );
```

**Arguments:**

- **bus**
  The bus number.

- **dev_func**
  The device or function ID. The device number is in bits 7 through 3, and the function number in bits 2 through 0.

- **offset**
  The offset into the configuration space. This must be aligned to a 16-bit boundary (that is 0, 2, 4, ..., 254 bytes).

- **count**
  The number of 16-bit values to write.

- **buff**
  A pointer to a buffer containing the data to be written into the configuration space.

**Library:**

```
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `pci_write_config16()` function writes individual 16-bit values to the configuration space of the specified device.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

**Returns:**

PCI_BAD_REGISTER_NUMBER

An invalid register offset was given.

PCI_BUFFER_TOO_SMALL

The `size` argument is greater than 50 words.

PCI_SUCCESS

The device or function was found.

-1 You haven’t called `pci_attach()`, or the call to it failed.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

**See also:**

`pci_attach()`, `pci_attach_device()`, `pci_detach()`, `pci_detach_device()`, `pci_find_class()`, `pci_find_device()`, `pci_present()`, `pci_read_config()`, `pci_read_config8()`.
pci_write_config16()

pci_read_config16(), pci_read_config32(), pci_rescan_bus(),
pci_write_config(), pci_write_config8(), pci_write_config32()
pci_write_config32()

Write 32-bit values to the configuration space of a device

Synopsis:

```c
#include <hw/pci.h>

int pci_write_config32( unsigned bus,
                        unsigned dev_func,
                        unsigned offset,
                        unsigned count,
                        char* buff );
```

Arguments:

- `bus` The bus number.
- `dev_func` The device or function ID. The device number is in bits 7 through 3, and the function number in bits 2 through 0.
- `offset` The register offset into the configuration space. This must be aligned to a 32-bit boundary (that is 0, 4, 8, ..., 252 bytes).
- `count` The number of 32-bit values to write.
- `buff` A pointer to a buffer containing the data to be written into the configuration space.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pci_write_config32()` function writes individual 32-bit values to the configuration space of the specified device.
You must successfully call `pci_attach()` before calling any of the other PCI functions.

**Returns:**

- `PCI_BAD_REGISTER_NUMBER`
  - An invalid register offset was given.

- `PCI_SUCCESS`
  - The device or function was found.

- `-1` You haven’t called `pci_attach()`, or the call to it failed.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pci_attach()`, `pci_attach_device()`, `pci_detach()`, `pciDetach_device()`, `pci_find_class()`, `pci_find_device()`, `pci_present()`, `pci_read_config()`, `pci_read_config8()`, `pci_read_config16()`, `pci_read_config32()`, `pci_rescan_bus()`, `pci_write_config()`, `pci_write_config8()`, `pci_write_config16()`
Synopsis:

```c
#include <stdio.h>

int pclose( FILE* stream );
```

Arguments:

- `stream`: The stream pointer for the pipe that you want to close, that you obtained by calling `popen()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pclose()` function closes the pipe associated with `stream`, and waits for the subprocess created by `popen()` to terminate.

Returns:

The termination status of the command language interpreter, or -1 if an error occurred (`errno` is set).

Errors:

- **EINTR**: The `pclose()` function was interrupted by a signal while waiting for the child process to terminate.
- **ECHILD**: The `pclose()` function was unable to obtain the termination status of the child process.
Examples:

See `popen()`.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`errno, popen(), pipe()`
Synopsis:

```c
#include <stdio.h>

void perror( const char *prefix );
```

Arguments:

- `prefix` NULL, or a string that you want to print before the error message.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `perror()` function prints the following to `stderr`:

- the given `prefix`, followed by " : "
- the error message returned by `strerror()` for the current value of `errno`
- a newline character.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    fp = fopen( "data.fil", "r" );
    if( fp == NULL ) {
        perror( "Unable to open file" );
        return EXIT_FAILURE;
    } return EXIT_SUCCESS;
}
```
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`errno, fprintf(), stderr, strerror()`
pipe()
Create a pipe

Synopsis:
#include <unistd.h>

int pipe( int fildes[2] );

Arguments:

fildes  An array where the function can store the file descriptors for
        the ends of the pipe.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The pipe() function creates a pipe (an unnamed FIFO) and places a
file descriptor for the read end of the pipe in fildes[0], and a file
descrptor for the write end of the pipe in fildes[1]. Their integer
values are the two lowest available at the time of the pipe() function
call. The O_NONBLOCK flag is cleared for both file descriptors. (You
can use fcntl() to set the O_NONBLOCK flag.)

You can write data to file descriptor fildes[1] and read it from file
descriptor fildes[0]. If you read from file descriptor fildes[0], it returns
the data written to fildes[1] on a first-in-first-out (FIFO) basis.

The pipe buffer is allocated by the pipe resource manager.

You typically use this function to connect standard utilities acting as
filters, passing the write end of the pipe to the data-producing process
as its STDOUT_FILENO, and the read end of the pipe to the
data-consuming process as its STDIN_FILENO (either via the
traditional fork(), dup2(), or exec*, or the more efficient spawn*
calls).

If successful, pipe() marks the st_fime, st_ctime, st_atime and
st_mtime fields of the pipe for updating.
pipe()

Returns:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (errno is set).</td>
</tr>
</tbody>
</table>

Errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMFILE</td>
<td>The calling process doesn’t have at least 2 unused file descriptors available.</td>
</tr>
<tr>
<td>ENFILE</td>
<td>The number of simultaneously open files in the system would exceed the configured limit.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>There’s insufficient space available to allocate the pipe buffer.</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>There’s no pipe manager running.</td>
</tr>
<tr>
<td>EROFS</td>
<td>The pipe pathname space is a read-only filesystem.</td>
</tr>
</tbody>
</table>

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fcntl(), nbacconnect(), open(), popen(), read(), write()

pipe in the Utilities Reference


**Synopsis:**

```c
#include <sys/poll.h>

int poll( struct pollfd fds*,
          nfds_t nfds,
          int timeout );
```

**Arguments:**

- `fds` The array of interest.
- `nfds` Number of elements in the `fds` array.
- `timeout` Timeout in milliseconds.

**Library:**

`libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `poll()` function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. The `pollfd` structure has the following components:

```
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

For each member of the array pointed to by `fds`, `poll()` examines the given file descriptor for the event(s) specified in `events`. The number of `pollfd` structures in the `fds` array is specified by `nfds`. The array’s members are `pollfd` structures within which `fd` specifies an open file descriptor, `events` and `revents` are bitmasks constructed by OR’ing a combination of the following event flags:
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLERR</td>
<td>An error has occurred on the device. This flag is valid only in the <code>revents</code> bitmask; it's ignored in the <code>events</code> member.</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>The device has been disconnected. This event and POLLOUT are mutually exclusive; a device can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNorm, POLLRDBAND, or POLLPRI are not mutually exclusive. If the remote end of a socket is closed, <code>poll()</code> indicates a POLLIN event rather than POLLHUP. This flag is valid only in the <code>revents</code> bitmask; it's ignored in the <code>events</code> member.</td>
</tr>
<tr>
<td>POLLIN</td>
<td>Data other than high-priority data may be read without blocking. This is equivalent to POLLRDNORM</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>The specified fd value is invalid. This flag is only valid in the <code>revents</code> member; it shall ignored in the <code>events</code> member.</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>Normal data may be written without blocking.</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>High-priority data may be read without blocking.</td>
</tr>
<tr>
<td>POLLRDBAND</td>
<td>Priority data may be read without blocking.</td>
</tr>
<tr>
<td>POLLRDNORM</td>
<td>Normal data may be read without blocking.</td>
</tr>
<tr>
<td>POLLWRBAND</td>
<td>Priority data may be written.</td>
</tr>
<tr>
<td>POLLWRNORM</td>
<td>Equivalent to POLLOUT.</td>
</tr>
</tbody>
</table>
The significance and semantics of normal, priority, and high-priority data are file- and device-specific.

If the value of \textit{fd} is less than 0, events are ignored; and \textit{revents} are set to 0 in that entry on return from \textit{poll}().

In each \texttt{pollfd} structure, \textit{poll}() clears the \texttt{revents} member, except that where the application requested a report on a condition by setting one of the bits of events listed above, \textit{poll}() sets the corresponding bit in \texttt{revents} if the requested condition is true. In addition, \textit{poll}() sets the \texttt{POLLHUP}, \texttt{POLLERR}, and \texttt{POLLNVAL} flag in \texttt{revents} if the condition is true, even if the application didn’t set the corresponding bit in events.

If none of the defined events occurs on any selected file descriptor, \textit{poll}() waits at least \textit{timeout} milliseconds for an event to occur on any of the selected file descriptors. If the value of \textit{timeout} is 0, \textit{poll}() returns immediately. If the value of \textit{timeout} is -1, \textit{poll}() blocks until a requested event occurs or until the call is interrupted.

The \textit{poll}() function isn’t affected by the O\_NONBLOCK flag.

The \textit{poll}() function reports regular files, terminal and pseudo-terminal devices, FIFOs, and pipes.

Regular files always poll TRUE for reading and writing.

A file descriptor for a socket that’s listening for connections indicates that it’s ready for reading, once connections are available. A file descriptor for a socket that connects asynchronously indicates that it’s ready for writing, once a connection has been established.

\textbf{Returns:}

\begin{itemize}
  \item \texttt{> 0} Total number of file descriptors that have been selected.
  \item \texttt{0} The call timed out, and no file descriptor has been selected.
  \item \texttt{-1} Failure, and \texttt{errno} is set.
\end{itemize}
Errors:

EAGAIN  The allocation of internal data structures failed but a subsequent request may succeed.

EINTR  A signal was caught during poll()

EFAULT  The \texttt{fds} argument pointed to a nonexistent portion of the calling process’s address space.

Examples:

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/poll.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

struct sockaddr_in sad;

void *
client(void *arg)
{
    int s;
    const char *p = "Some data\n";

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return NULL;
    }

    if (connect(s, (struct sockaddr *)&sad, sizeof(sad)) == -1) {
        perror("connect");
        return NULL;
    }

    write(s, p, strlen(p));
    close(s);

    return NULL;
}

int
main(void)
```
```c
{
    struct pollfd fds;
    int s = -1, s2 = -1, done_accept = 0, oflags, ret;
    char buf[100];

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return 1;
    }

    sad.sin_family = AF_INET;
    sad.sin_len = sizeof(sad);
    sad.sin_addr.s_addr = inet_addr("127.0.0.1");
    sad.sin_port = htons(1234);

    fds.fd = s;
    fds.events = POLLRDNORM;

    oflags = fcntl(s, F_GETFL);
    oflags |= O_NONBLOCK;
    fcntl(s, F_SETFL, oflags);

    if (bind(s, (struct sockaddr *)&sad, sizeof(sad)) == -1) {
        perror("bind");
        return 1;
    }

    listen(s, 5);

    if ((ret = pthread_create(NULL, NULL, client, NULL)) != EOK) {
        fprintf(stderr, "pthread_create: %s\n", strerror(ret));
        return 1;
    }

    for (;;) {
        if ((ret = poll(&fds, 1, -1)) == -1) {
            perror("poll");
            break;
        } else if (ret != 1 || (fds.revents & POLLRDNORM) == 0) {
            break;
        }

        if (done_accept) {
            if ((ret = read(s2, buf, sizeof(buf))) <= 0) { break; }
        }

        printf("%s", buf);
    }
}
```
else {
    if ((s2 = accept(s, NULL, 0)) == -1) {
        perror("accept");
        break;
    }

    fds.fd = s2;
    done_accept = 1;
}
}

close(s);
close(s2);
return 0;

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Not all managers support POLLPRI, POLLPRI, POLLERR, and POLLHUP.

See also:

read(), select(), write()

<sys/poll.h>
Synopsis:

```
#include <stdio.h>

FILE* popen( const char* command,
             const char* mode );
```

Arguments:

- `command`  The command that you want to execute.
- `mode` The I/O mode for the pipe, which must be "r" or "w"; see below.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `popen()` function executes the command specified by `command` and creates a pipe between the calling process and the executed command.

Depending on the `mode` argument, you can use the returned stream pointer to read from or write to the pipe.

The executed command has the same environment as its parents. The command is started as follows:

```
spawnlp (P_NOWAIT, shell_command, shell_command,
         "-c", command, (char*) NULL );
```

where `shell_command` is the command specified by the `SHELL` environment variable (if it exists), or the `sh` utility.
The *mode* argument to *popen()* is a string that specifies an I/O mode for the pipe:

- If *mode* is '*r*', then when the child process is started:
  - Its file descriptor, STDOUT_FILENO, is the writable end of the pipe.
  - The `fileno(stream)` in the calling process is the readable end of the pipe, where `stream` is the stream pointer returned by *popen()*.

- If *mode* is '*w*', then when the child process is started:
  - Its file descriptor, STDIN_FILENO, is the readable end of the pipe.
  - The `fileno(stream)` in the calling process is the writable end of the pipe, where `stream` is the stream pointer return by *popen()*.

- If *mode* is any other value, the result is undefined.

Use *pclose()* to close a stream that you used *popen()* to open.

**Returns:**

A non-NULL stream pointer on successful completion. If *popen()* is unable to create either the pipe or the subprocess, it returns a NULL stream pointer and sets *errno*.

**Errors:**

**EINVAL** \(\text{The } \text{mode} \text{ argument is invalid.}\)

**ENOSYS** \(\text{There’s no pipe manager running.}\)

The *popen()* function may also set *errno* values as described by the *pipe()* and *spawnl()* functions.
popen()

Examples:

/*
 * upper: executes a given program, converting all input
 *        to upper case.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <limits.h>

char buffer[_POSIX_ARG_MAX];

int main( int argc, char** argv )
{
    int i;
    int c;
    FILE* f;

    for( i = 1; i < argc; i++ ) {
        strcat( buffer, argv[i] );
        strcat( buffer, " " );
    }
    if( ( f = popen( buffer, "w" ) ) == NULL ) {
        perror( "popen" );
        return EXIT_FAILURE;
    }
    while( ( c = getchar() ) != EOF ) {
        if( islower( c ) )
            c = toupper( c );
        putc( c, f );
    }
    pclose( f );
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

Cancellation point  Yes
Interrupt handler    No

continued...
popen()

Safety

- Signal handler: No
- Thread: No

See also:

`errno`, `pclose()`, `pipe()`, `spawnlp()`
Get the offset and length of a mapped typed memory block

Synopsis:

```c
#include <sys/mman.h>

int posix_mem_offset(const void *restrict addr, 
                     size_t len, 
                     off_t *restrict off, 
                     size_t *restrict contig_len, 
                     int *restrict fildes);

int posix_mem_offset64(const void *restrict addr, 
                        size_t len, 
                        off64_t *restrict off, 
                        size_t *restrict contig_len, 
                        int *restrict fildes);
```

Arguments:

- `addr` The address that the memory block is currently mapped at.
- `contig_len` A pointer to either `len`, or the length of the largest contiguous block of typed memory that’s currently mapped to the calling process starting at `addr`, whichever is smaller.
- `len` The length of the block of memory object that you want the offset of.
- `off` A pointer to a location where the function can store the offset (or location) within the memory block.
- `fildes` A pointer to a location where the function can store the file descriptor for the typed memory object.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**Description:**

The `posix_mem_offset()` function sets the variable pointed to by `off` to the offset (or location), within a typed memory object, of the memory block currently mapped at `addr`.

The `posix_mem_offset()` function uses a variable to return the descriptor that establishes the mapping containing `addr`. This variable is pointed to by `fildes`; its value is -1 when the descriptor closes after the mapping is established.

The `len` argument specifies the length of the block of memory you want the offset for. On return, the value pointed to by `contig_len` is either `len`, or the length of the largest contiguous block of typed memory that’s currently mapped to the calling process starting at `addr`, whichever is smaller.

If the `off` and `contig_len` values obtained from calling `posix_mem_offset()` are used in a call to `mmap()` with a file descriptor that refers to the same memory pool as `fildes` (either through the same port or through a different port), the typed memory area that is mapped is exactly the same area that was mapped at `addr` of the process that called `posix_mem_offset()`. Note that neither of the two flags (such as the POSIX_TYPED_MEMORY_ALLOCATE or the POSIX_TYPED_MEM_ALLOCATE_CONTIG) did open the file descriptor.

**Returns:**

- 0    Success.
- EACCES The process hasn’t mapped a memory object at the given address `addr`.

**Classification:**

`posix_mem_offset()` is POSIX 1003.1 TYM; `posix_mem_offset64()` is for large-file support.
posix_mem_offset(), posix_mem_offset64() © 2007, QNX
Software Systems GmbH & Co. KG.

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

posix_TYPED_mem_get_info(), posix_TYPED_mem_open()
Synopsis:

```c
#include <stdlib.h>

int posix_memalign( void ** memptr,
    size_t alignment,
    size_t size );
```

Arguments:

- **memptr**: A pointer to a location where `posix_memalign()` can store a pointer the memory.
- **alignment**: The alignment to use for the memory. This must be a multiple of `size(void*)`.
- **size**: The size, in bytes, of the block to allocate.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_memalign()` function allocates `size` bytes aligned on a boundary specified by `alignment`. It returns a pointer to the allocated memory in `memptr`.

The buffer allocated by `posix_memalign()` is contiguous in virtual address space, but not physical memory. Since some platforms don’t allocate memory in 4K page sizes, you shouldn’t assume that the memory allocated will be physically contiguous if you specify a size of 4K or less.

You can obtain the physical address of the start of the buffer using `mem_offset()` with `fd=NOFD`. 

September 10, 2007  C Library — P to R  2083
Returns:

0       Success.
-1      An error occurred (errno is set).

Errors:

EINVAL  The value of alignment isn’t a multiple of size( void * ).
ENOMEM  There’s insufficient memory available with the requested alignment.

Classification:

POSIX 1003.1 ADV

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, free(), malloc(), memalign()
posix_typed_mem_get_info()

Query a typed memory object to obtain the amount of memory available.

Synopsis:

```c
#include <sys/mman.h>

int posix_typed_mem_get_info(
    int fildes,
    struct posix_typed_mem_info *info);
```

Arguments:

- `fildes` The file description for the typed memory object that you want to query.
- `info` A pointer to a `posix_typed_mem_info` structure where the function can store the information.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `posix_typed_mem_get_info()` function queries the typed memory object to obtain the amount of memory currently available.

The `posix_typed_mem_get_info()` function allows an application to immediately determine the amount of available memory. This is particularly important for typed memory objects that may in some cases be scarce resources. The `posix_typed_mem_get_info()` function stores, in the `posix_tmi_length` field of the `posix_typed_mem_info` structure pointed to by `info`, the maximum length that may be successfully allocated by the typed memory object designated by `fildes`. Note that when a typed memory pool is a shared resource, some form of mutual-exclusion or synchronization may be required while typed memory is being queried and allocated to prevent race conditions.
The maximum length is dynamic; it’s valid only while the current mapping of the corresponding typed memory pool remains unchanged. This maximum length takes into account the flag POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG specified when the typed memory object represented by fildes was opened.

If fildes represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE flag nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag specified, the returned value of info->posix_tmi_length is 0.

This function was added in the QNX Neutrino Core OS 6.3.2.

**Returns:**

<table>
<thead>
<tr>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The fildes argument isn’t a valid open file descriptor.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>The fildes isn’t connected to a memory object supported by this function.</td>
</tr>
</tbody>
</table>

**Classification:**

<table>
<thead>
<tr>
<th>POSIX</th>
<th>1003.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`posix_mem_offset()`, `posix_TYPED_mem_open()`
**Synopsis:**
```c
c#include <sys/mman.h>

int posix_typed_mem_open(const char * name,  
                         int oflag, 
                         intt * tflag);
```

**Arguments:**
- **name** A pointer to the string that specifies the typed memory object.
- **oflag** The access mode to use; one of:
  - O_RDONLY Open for read access only.
  - O_WRONLY Open for write access only.
  - O_RDWR Open for read or write access.
- **tflag** Flag that determine how the typed memory object behaves when mapped; at most one of:
  - POSIX_TYPED_MEM_ALLOCATE
    Allocate on `mmap()`.
  - POSIX_TYPED_MEM_ALLOCATE_CONTIG
    Allocate contiguously on `mmap()`.
  - POSIX_TYPED_MEM_MAP_ALLOCATABLE
    Map on `mmap()` without affecting the availability for allocation.

**Library:**
- libc

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `posix_typed_mem_open()` function establishes a connection between the typed memory object specified by the `name` and a file descriptor. It creates an open file description that refers to the typed memory object and a file descriptor that refers to the open file description.

The file descriptor is used by other functions to refer to that typed memory object.

The QNX Neutrino refers to the same typed memory object whether the `name` begins with the slash character or not, i.e. `/sysram` and `sysram` refer to the same typed memory object.

When `tflag` is set to `POSIX_TYPED_MEM_ALLOCATE`, any subsequent call to `mmap()` that uses the returned file descriptor shall result in the allocation and mapping of typed memory from the specified typed memory pool. The allocated memory will be either of the previously unallocated contiguous area of the typed memory pool or previously unallocated (several) noncontiguous areas (mapped to a contiguous portion of the process address space).

When `tflag` is set to `POSIX_TYPED_MEM_ALLOCATE_CONTIG`, any subsequent call to `mmap()` that uses the returned file descriptor results in the allocation and mapping of a previously unallocated single contiguous area of the typed memory pool (also mapped to a contiguous portion of the process address space).

If `tflag` isn’t set to either `POSIX_TYPED_MEM_ALLOCATE` or `POSIX_TYPED_MEM_ALLOCATE_CONTIG`, any subsequent call to `mmap()` that uses the returned file descriptor maps an application-chosen area from the specified typed memory pool. Note that this mapped area becomes unavailable for allocation until unmapped by all processes.

When `tflag` is set to `POSIX_TYPED_MEM_MAP_ALLOCATABLE`, any subsequent call to `mmap()` using the returned file descriptor maps an application-chosen area from the specified typed memory pool without an effect on the availability of that area for allocation; that is, mapping such an object leaves each byte of the mapped area...
unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping.

If successful, `posix_typed_mem_open()` returns a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file descriptor is new, and therefore the file descriptor won’t share it with any other processes. The FD_CLOEXEC file descriptor flag associated with the new file descriptor is cleared.

The behavior of `msync()`, `ftruncate()`, and all file operations other than `mmap()`, `posix_mem_offset()`, `posix_typed_mem_get_info()`, `fstat()`, `dup()`, `dup2()`, and `close()` is as follows (see the next paragraph) when passed a file descriptor connected to a typed memory object by this function. The file status flags of the open file description is set according to the value of `oflag`.

The `msync()` function performs the QNX extension, i.e. it flushes and/or invalidates data and or instruction cache for the specified memory region.

All other functions return error with ENOSYS.

---

This function was added in the QNX Neutrino Core OS 6.3.2.

---

**Returns:**

A nonnegative integer that represents the lowest-numbered unused file descriptor, or -1 if an error occurred (`errno` is set).

**Errors:**

- **EACCES** The typed memory object exists and the permissions specified by `oflag` are denied.
- **EINVAL** The `posix_typed_mem_open()` API was interrupted by a signal.
- **EINVAL** The flags specified in `tflag` are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE,
**posixTypedMemOpen()**

POSIX_TYPED_MEM_ALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).

**EMFILE**  
Too many file descriptors are currently in use by this process.

**ENOMEM**  
The length of the name argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

**ENFILE**  
Too many file descriptors are currently open in the system.

**ENOENT**  
The named typed memory object doesn’t exist.

**EPERM**  
The caller lacks the superuser capability to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument *tflag*.

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`posixTypedMemGetInfo()`, `posixMemOffset()`
pow(), powf()

Raise a number to a given power

Synopsis:

```c
#include <math.h>

double pow( double x,
            double y );

float powf( float x,
            float y );
```

Arguments:

- `x`: The number you want to raise.
- `y`: The power you want to raise the number to.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `pow()` and `powf()` functions compute $x$ raised to the power of $y$.

A domain error occurs if $x = 0$, and $y \leq 0$, or if $x$ is negative, and $y$ isn’t an integer. A range error may also occur.

Returns:

- The value of $x^y$.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
    printf( "%.6lf\n", pow( 1.5, 2.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
2.755676
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, exp(), log(), sqrt()
**pread(), pread64()**

*Read from a file without moving the file pointer*

### Synopsis:

```c
#include <unistd.h>

ssize_t pread(int filedes,
               void *buff,
               size_t nbytes,
               off_t offset);

ssize_t pread64(int filedes,
                 void *buff,
                 size_t nbytes,
                 off64_t offset);
```

### Arguments:

- **filedes**: The descriptor of the file that you want to read from.
- **buff**: A pointer to a buffer where the function can store the data that it reads.
- **nbytes**: The number of bytes that you want to read.
- **offset**: The desired position inside the file.

### Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `pread()` function performs the same action as `read()`, except that it reads from a given position in the file without changing the file pointer.

The `pread()` function reads up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread()` on a file that’s incapable of seeking results in an error.

The `pread64()` function is a 64-bit version of `pread()`.
Returns:

The number of bytes actually read, or -1 if an error occurred (errno is set).

Errors:

EAGAIN  The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the read operation.

EBADF   The file descriptor, fildes, isn’t a valid file descriptor open for reading.

EINTR   The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file does not report partial transfers.

EIO     A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSYS  The pread() function isn’t implemented for the filesystem specified by filedes.

Classification:

pread() is POSIX 1003.1 XSI; pread64() is Large-file support

See also:

close(), creat(), dup(), dup2(), errno, fcntl(), lseek(), open(), pipe(), pwrite(), read(), readblock(), readv(), select(), write(), writeblock(), writev()
printf() — Write formatted output to stdout

Synopsis:

```c
#include <stdio.h>

int printf( const char * format, ...
            );
```

Arguments:

- `format` A string that controls the format of the output, as described below. The formatting string determines what additional arguments you need to provide.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `printf()` function writes output to the `stdout` stream, under control of the argument `format`.

If the format string contains invalid multibyte characters, processing stops, and the rest of the format string, including the `%` characters, is printed. This can happen, for example, if you specify international characters, accents, and diacritical marks using ISO 8859-1 instead of UTF-8. If you call:

```c
setlocale( LC_CTYPE, "C-TRADITIONAL" );
```

before calling `printf()`, the locale switches multibyte processing from UTF-8 to 1-to-1, and `printf()` safely transfers the misformed multibyte characters.
Printf

Format Arguments

If there are leftover arguments after processing format, they’re ignored.

The printf() family of functions allows for language-dependent radix characters. The default character is “.”, but is controlled by LC_NUMERIC and setlocale().

Format control string

The format control string consists of:

multibyte characters
These are copied to the output stream exactly as they occur in the format string. An ordinary character in the format string is any character, other than a percent character (%), that isn’t part of a conversion specifier.

conversion specifiers
These cause argument values to be written as they’re encountered during the processing of the format string. A conversion specifier is a sequence of characters in the format string that begins with “%” and is followed by:

- zero or more format control flags that can modify the final effect of the format directive
- an optional decimal integer, or an asterisk (*), that specifies a minimum field width to be reserved for the formatted item
- an optional precision specification in the form of a period (.), followed by an optional decimal integer or an asterisk (*)
- an optional type length specification, one of: h, hh, j, l, ll, L, t or z.
- a character that specifies the type of conversion to be performed. See below.
Format control flags

The valid format control flags are:

- Left-justify the formatted item within the field; normally, items are right-justified.

+ Always start a signed, positive object with a plus character (+); normally, only negative items begin with a sign.

space Always start a signed, positive object with a space character; if both + and a space are specified, + overrides the space.

# Use an alternate conversion form:

- For o (unsigned octal) conversions, increment the precision, if necessary, so that the first digit is 0.
- For x or X (unsigned hexadecimal) conversions, prepend a nonzero value with 0x or 0X.
- For e, E, f, g, or G (any floating-point) conversions, always include a decimal-point character in the result, even if no digits follow it; normally, a decimal-point character appears in the result only if there is a digit to follow it.
- In addition, for g or G conversions, don’t remove trailing zeros from the result.

0 (zero) Use leading zeros to pad the field width for d, i, o, u, x, X, e, E, f, g and G conversions. The “-” flag overrides the this flag.

Field width

If you don’t specify a field width, or if the given value is less than the number of characters in the converted value (subject to any precision value), a field of sufficient width to contain the converted value is used.
If the converted value has fewer characters than specified by the field width, the value is padded on the left (or right, subject to the left-justification flag) with spaces or zero characters (0). If the field width begins with a zero, the value is padded with zeros; otherwise, the value is padded with spaces.

If the field width is * (or 0*), a value of type int from the argument list is used (before a precision argument or a conversion argument) as the minimum field width. A negative field width value is interpreted as a left-justification flag, followed by a positive field width.

**Precision specifier**

As with the field width specifier, a precision specifier of * causes a value of type int from the argument list to be used as the precision specifier. If you give a precision specifier of *, but there’s no precision value in the argument list, a precision of 0 is used.

The precision value affects the following conversions:

- For d, i, o, u, x and X (integer) conversions, the precision specifies the minimum number of digits to appear.
- For e, E and f (fixed-precision, floating-point) conversions, the precision specifies the number of digits to appear after the decimal-point character.
- For g and G (variable-precision, floating-point) conversions, the precision specifies the maximum number of significant digits to appear.
- For s (string) conversions, the precision specifies the maximum number of characters to appear.

**Type length specifier**

A type length specifier affects the conversion as follows:

- h causes a d, i, o, u, x or X (integer) format conversion to treat the argument as a short or unsigned short argument.
Note that, although the argument may have been promoted to an int as part of the function call, the value is converted to the smaller type before it’s formatted.

- h causes an n (converted length assignment) operation to assign the converted length to an object of type short.
- hh is similar to h, but treats the argument as a signed char or an unsigned char.
- j causes a d, i, o, u, x, X format conversion to process a intmax_t or uintmax_t.
- j causes an n format conversion to process an intmax_t.
- L causes an a, A, e, E, f, g, G (double) format conversion to process a long double argument.
- l ("el") causes a c format conversion to process a wchar_t argument.
- l ("el") causes an s format conversion to process a wchar_t argument.
- l ("el") causes a d, i, o, u, x, or X (integer) format conversion to process a long or unsigned long argument.
- l ("el") causes an n (converted length assignment) operation to assign the converted length to an object of type long.
- ll (double “el”) causes a d, i, o, u, x, or X (integer) format conversion to assign the converted value to an object of type long long or unsigned long long.
- ll (double “el”) causes an n (converted length assignment) operation to assign the number of characters that have been read to an object of type long long.
- t causes a d, i, o, u, x, X format conversion to process a ptrdiff_t or the corresponding unsigned type argument.
- `t` causes an `n` format conversion to process a `ptrdiff_t` argument.

- `z` causes a `d`, `i`, `o`, `u`, `x`, `X` format conversion to process a `size_t` argument.

- `z` causes a `n` format conversion to process a `size_t` argument.

### Conversion type specifiers

The valid conversion type specifiers are:

- **a, A** Convert an argument of type `double` in the style `[-]0xh.hhhh p1d`, where there’s one nonzero hexadecimal digit before the decimal point. The number of hexadecimal digits after the decimal point is equal to the precision. If the precision is missing and FLT_RADIX is a power of 2, then the precision is sufficient for an exact representation. If the precision is zero and you don’t specify the `#` flag, no decimal point is shown.

  The `a` conversion uses the letters `abcdef` and produces `x` and `p`; the `A` conversion `ABCDEF`, `X` and `P`. The exponent always has one digit, even if it’s 0, and no more digits than necessary. The values for infinity or NaN are converted in the style of an `f` or `F`.

- **c** Convert an argument of type `int` into a value of type `unsigned char` and write the corresponding ASCII character code to the output stream.

  An `l` ("el") qualifier causes a `wint_t` argument to be converted as if by an `ls` conversion into a `wchar_t`, the first element being the `wint_t` and the second being a null wide character.

- **d, i** Convert an argument of type `int` into a signed decimal notation and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.
e, E  Convert an argument of type double into a decimal notation in the form [-] d.ddd e[+ | -]dd. The leading sign appears (subject to the format control flags) only if the argument is negative.

If the argument is nonzero, the digit before the decimal-point character is nonzero. The precision is used as the number of digits following the decimal-point character. If you don’t specify the precision, a default precision of six is used. If the precision is 0, the decimal-point character is suppressed. The value is rounded to the appropriate number of digits.

The exponent sign and the exponent (that indicates the power of ten by which the decimal fraction is multiplied) are always produced. The exponent is at least two digits long and has only as many additional digits as necessary to represent it. If the value is zero, the exponent is zero.

For E conversions, the exponent begins with the character E, rather than e.

The arguments infinity or NaN are converted in the style of the f or F conversion specifiers.

f, F  Convert an argument of type double into a decimal notation in the form [-] ddd.ddd with the number of digits after the decimal point being equal to the precision specification. The leading sign appears (subject to the format control flags) only if the argument is negative.

The precision is used as the number of digits following the decimal-point character. If you don’t specify the precision, a default precision of six is used. If the precision is 0, the decimal-point character is suppressed; otherwise, at least one digit is produced before the decimal-point character. The value is rounded to the appropriate number of digits.

An argument of type double that represents infinity or NaN is converted to [-] inf or [-] nan. The F specifier produces [-] INF or [-] NAN.
printf()

**g, G** Convert an argument of type `double` using either the `e` or `f` (or `E`, for a `G` conversion) style of conversion, depending on the value of the argument. In either case, the precision specifies the number of significant digits that are contained in the result. The `e` style conversion is used only if the exponent from such a conversion would be less than -4 or greater than the precision. Trailing zeros are removed from the result, and a decimal-point character only appears if it is followed by a digit.

Arguments representing infinity or NaN are converted in the style of the `f` or `F` conversion specifiers.

**n** Assign the number of characters that have been written to the output stream to the integer pointed to by the argument. No output is produced.

**o** Convert an argument of type `unsigned` into an unsigned octal notation, and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.

**p** Convert an argument of type `void *` into a value of type `int`, and format the value as for a hexadecimal (`x`) conversion.

**s** Write the characters from the string specified by an argument of type `char *`, up to, but not including the terminating NUL character (`\0`), to the output stream. If you specify a precision, no more than that many characters are written.

If you use an `l` ("el") qualifier, the argument is interpreted as a pointer to a `wchar_t` array, and each wide character, including the terminating NUL, is converted as if by a call to `wcrtomb()`. The terminating NUL is written only if you don’t specify the precision, or if you specify the precision and the length of the character sequence is less than the precision.

**u** Convert an argument of type `unsigned` into an unsigned decimal notation, and write it to the output stream. The
default precision is 1, but if more digits are required, leading zeros are added.

\texttt{x, X} \hspace{1em} \texttt{Convert an argument of type unsigned into an unsigned hexadecimal notation, and write it to the output stream. The default precision is 1, but if more digits are required, leading zeros are added.}

Hexadecimal notation uses the digits 0 through 9 and the characters a through f or A through F for \texttt{x} or \texttt{X} conversions, respectively, as the hexadecimal digits. Subject to the alternate-form control flag, 0x or 0X is prepended to the output.

\% \hspace{1em} \texttt{Print a \% character (The entire specification is \%).}

Any other conversion type specifier character, including another percent character (\%), is written to the output stream with no special interpretation.

The arguments must correspond with the conversion type specifiers, left to right in the string; otherwise, indeterminate results will occur.

If the value corresponding to a floating-point specifier is infinity, or not a number (NAN), then the output will be \texttt{inf} or -\texttt{inf} for infinity, and \texttt{nan} or -\texttt{nan} for NANs.

For example, a specifier of the form \%8.\ast f defines a field to be at least 8 characters wide, and gets the next argument for the precision to be used in the conversion.

\textbf{Returns:}

The number of characters written, excluding the terminating NULL, or a negative number if an error occurred (\texttt{errno} is set).

\textbf{Examples:}

\begin{verbatim}
#include <stdio.h>
#include <stdlib.h>

int main( void )
{

\end{verbatim}
char *weekday, *month;

weekday = "Saturday";
month = "April";
printf("%s, %s %d, %d\n", weekday, month, 10, 1999);
printf("f1 = %8.4f f2 = %10.2E x = %#08x i = %d\n",
       23.45, 3141.5926, 0x1db, -1);
return EXIT_SUCCESS;
}

produces the output:

Saturday, April 10, 1999
f1 = 23.4500 f2 = 3.14E+003 x = 0x0001db i = -1

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fprintf(), fwprintf(), snprintf(), sprintf(), swprintf(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
procmgr_daemon()  Run a process in the background

Synopsis:

```
#include <sys/procmgr.h>

int procmgr_daemon(int status,
                    unsigned flags);
```

Arguments:

- `status` The status that you want to return to the parent process.
- `flags` The flags currently defined (in `<sys/procmgr.h>`) are:

  - `PROCMGR_DAEMON_NOCHDIR` — unless this flag is set, `procmgr_daemon()` changes the current working directory to the root “/”.
  - `PROCMGR_DAEMON_NOCLOSE` — unless this flag is set, `procmgr_daemon()` closes all file descriptors other than standard input, standard output and standard error.
  - `PROCMGR_DAEMON_NODEVNULL` — unless this flag is set, `procmgr_daemon()` redirects standard input, standard output and standard error to `/dev/null`.
  - `PROCMGR_DAEMON_KEEPUMASK` — unless this flag is set, `procmgr_daemon()` sets the umask to 0 (zero).

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `procmgr_daemon()` function lets programs detach themselves from the controlling terminal and run in the background as system daemons. This also puts the caller into session 1.

The argument `status` is returned to the parent process as if `exit()` were called; the returned value is normally `EXIT_SUCCESS`. 
The data in the siginfo_t structure for the SIGCHLD signal that the parent receives isn’t useful in this case.

Returns:

A nonnegative integer, or -1 if an error occurs.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

demond(), exit(), procmgr_event_notify(), procmgr_event_trigger(), procmgr_guardian(), procmgr_session()
procmgr_event_notify()  © 2007, QNX Software Systems GmbH & Co. KG.
Ask to be notified of system-wide events

Synopsis:

```c
#include <sys/procmgr.h>

int procmgr_event_notify
    ( unsigned flags,
      const struct sigevent * event );
```

Arguments:

- **flags** Flags currently defined in `<sys/procmgr.h>` are:
  - `PROCMGR_EVENT_DAEMON_DEATH` — notify the caller when any process in session 1 dies. This is most useful for watching for the death of daemon processes that use `procmgr_daemon()` to put themselves in session 1 as well as close and redirect file descriptors. As a result of this closing and redirecting, the death of daemons are difficult to detect otherwise.
  - `PROCMGR_EVENT_PATHSPACE` — notify the caller when a resource manager adds or removes an entry (i.e. mountpoint) to or from the pathname space. This is generally associated with resource manager calls to `resmgr_attach()` and `resmgr_detach()`. Terminating a resource manager process also generates this event if the mountpoints have not been detached.
  - `PROCMGR_EVENT_SYNC` — notify the caller of any calls to `sync()` the filesystems.

Setting `flags` to 0 (zero) unarms the event.

Notification is via the given event, so no information is provided as to which process died. Once you’ve received the event, you’ll need to do something else to find out if processes you care about had died. You can do this by walking through the list of all processes, looking for specific process IDs or process names. If you don’t find one, then it has died. The sample code below demonstrates how this can be done.
**procmgr_event_notify()**

`event` A pointer to a `sigevent` structure that specifies how you want to notified.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `procmgr_event_notify()` function requests that the process manager notify the caller of the system-wide events identified by the given `flags`. A process may have only one notification request active at a time.

**Returns:**

-1 on error; any other value indicates success.

**Examples:**

```c
/*
 * This demonstrates procmgr_event_notify() with the
 * PROCMDR_DAEMON_DEATH flag. This flag allows you to
 * be notified if any process in session 1 dies.
 * Daemons are processes that do things that make
 * their death hard to detect (they become daemons by calling
 * procmgr_daemon()). One of the things that happens is that
 * daemons end up in session 1. Hence, the usefulness of the
 * PROCMDR_DAEMON_DEATH flag.
 *
 * When you are notified, you're not told who died.
 * It's up to you to know who should be running. Once notified,
 * you could then walk through the list of which processes are
 * still running and see if all the expected processes are still
 * running. If you know the process id of the processes you
 * are watching out for then this is easiest. If you don't know
 * the process id then your next option may be by process name.
 * The code below does a lookup by process name.
 */

#include <devctl.h>
#include <dirent.h>
#include <errno.h>
```

September 10, 2007
#include <fcntl.h>
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <sys/procmgr.h>

static int check_if_running( char *process );

#define DAEMON_DIED_CODE (_PULSE_CODE_MINAVAIL)

struct dinfo_s {
    procfs_debuginfo info;
    char pathbuffer[PATH_MAX];
};

int main( int argc, char **argv )
{
    char *daemon_to_watch;
    int chid, coid, rcvid;
    struct sigevent event;
    struct _pulse msg;

    if (argc != 2) {
        printf( "use: %s process_to_watch_for\n", argv[0] );
        exit( EXIT_FAILURE );
    }

    daemon_to_watch = argv[1]; /* the process to watch for */

    chid = ChannelCreate( 0 );
    coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );
    SIGEV_PULSE_INIT( &event, coid, SIGEV_PULSE_Prio_Inherit, DAEMON_DIED_CODE, 0 );

    /*
     * Ask to be notified via a pulse whenever a
     * daemon process dies
     */

    if (procmgr_event_notify( PROCMGR_EVENT_DAEMON_DEATH, &event ) == -1) {
        fprintf( stderr, "procmgr_event_notify() failed" );
        exit( EXIT_FAILURE );
    }

    while (1) {

rcvid = MsgReceive( chid, &msg, sizeof(msg), NULL );
if (rcvid != 0) {
    /* not a pulse; could be an unexpected message or
    error */
    exit( EXIT_FAILURE );
}
if (check_if_running( daemon_to_watch ) == 0)
    printf( "%s is no longer running\n", daemon_to_watch);
return 0;

static int
check_if_running( char *process )
{
    DIR *dirp;
    struct dirent *dire;
    char buffer[20];
    int fd, status;
    pid_t pid;
    struct dinfo_s dinfo;

    if ((dirp = opendir( "/proc" )) == NULL) {
        perror( "Could not open '/proc'" );
        return -1;
    }
    while (1) {
        if ((dire = readdir( dirp )) == NULL)
            break;
        if (isdigit( dire->d_name[0] )) {
            pid = strtoul( dire->d_name, NULL, 0 );
            sprintf( buffer, "/proc/%d/as", pid );
            if ((fd = open( buffer, O_RDONLY )) != NULL) {
                status = devctl( fd, DCMD_PROC_MAPDEBUG_BASE,
                                &dinfo, sizeof(dinfo), NULL );
                if (status == EOK) {
                    if (!strcmp( process,
                                  basename( dinfo.info.path ) ))
                        { closedir (dirp);

                    /* You should close fd to prevent memory leaking */
                    close(fd);

procmgr_event_notify()

    return 1;
  }
  } /* else some errors are expected, e.g. procnto
    has no MAPDEBUG info and there is a timing
    issue with getting info on the process
    that died, ignore errors */
  close( fd );
}
}
closedir( dirp );
return 0;

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

procmgr_daemon(), procmgr_event_trigger(), _pulse, sigevent
procmgr_event_trigger()

Trigger a system-wide event

Synopsis:

```c
#include <sys/procmgr.h>

int procmgr_event_trigger( unsigned flags );
```

Arguments:

- **flags**
  
  The type of event that you want to trigger (defined in `<sys/procmgr.h>`):

  - PROCMGR_EVENT_SYNC — notify filesystems to `sync()`.

Library:

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `procmgr_event_trigger()` triggers a system-wide event. The event is sent to all processes that requested (via `procmgr_event_notify()`) to be notified of the event identified by `flags`.

Returns:

- `-1` on error; any other value indicates success.

Examples:

```c
#include <sys/procmgr.h>

int main ( void )
{
    procmgr_event_trigger( PROCMGR_EVENT_SYNC );
}
```
procmgr_event_trigger()  © 2007, QNX Software Systems GmbH & Co. KG.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
procmgr_event_notify(), sync()
Let a daemon process take over as a parent

**Synopsis:**

```c
#include <sys/procmgr.h>

pid_t procmgr_guardian( pid_t pid );
```

**Arguments:**

- `pid` The ID of the child process that you want to become the guardian of the calling process’s other children.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The function `procmgr_guardian()` allows a daemon process to declare a child process to take over as parent to its children in the event of its death:

```
A (parent)  -- A dies -- C (parent)
  |                       |
 B           C (guardian)  B
```

*Specifying a guardian for child processes.*
Returns:

-1 on error; any other value on success.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <spawn.h>
#include <sys/procmgr.h>
#include <sys/wait.h>

pid_t child = -1;
pid_t guardian = -1;

/*
 * Build a list of the currently running children
 */
void check_children(void) {
    if(child > 0) {
        if(kill(child, 0) == -1) {
            child = -1;
        }
    }
    if(guardian > 0) {
        if(kill(guardian, 0) == -1) {
            guardian = -1;
        }
    }
}

void start_needed_children(void) {
    if(guardian == -1) {
        /* Make a child that will just sit around 
         * and wait for parent to die */
        while((guardian = fork()) == 0) {
            pid_t parent = getppid();
            /* Wait for parent to die.... */
            fprintf(stderr, "guardian %d waiting on parent %d\n",
                    getpid(), parent);
            while(waitpid(parent, 0, 0) != parent);
            /* Then loop around and take over */
        }
        if(guardian == -1) {
            fprintf(stderr, "Unable to start guardian\n");
        } else {
```
/* Declare the child a guardian */
procmgr_guardian(guardian);
}

if(child == -1) {
    static char *args[] = { "sleep", "1000000", 0 };
    if((child = spawnp("sleep", 0, 0, 0, args, 0)) == -1) {
        fprintf(stderr, "Couldn’t start child\n");
        child = 0; /* don’t try again */
    }
}

int main(int argc, char *argv[]) {
    fprintf(stderr, "parent %d checking children\n", getpid());
    do {
        fprintf(stderr, "checking children\n");

        /* Learn about the newly adopted children */
        check_children();

        /* If anyone is missing, start them */
        start_needed_children();
        } while(wait(0)); /* Then wait for someone to die... */
    return EXIT_SUCCESS;
}

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`procmgr_daemon()`, `procmgr_event_notify()`,
`procmgr_event_trigger()`
Synopsis:

```c
#include <sys/procmgr.h>

int procmgr_session( uint32_t nd, pid_t sid, int id, unsigned event);
```

Arguments:

The interpretation of the arguments depends on the `event`.

- **nd**: A node descriptor.
- **sid**: A session ID.
- **id**: A file descriptor, process group, or signal, depending on the event.
- **event**: The event; one of:
  - `PROCMGR_SESSION_TCSETSID`
  - `PROCMGR_SESSION_SETSID`
  - `PROCMGR_SESSION_SETPGRP`
  - `PROCMGR_SESSION_SIGNAL_PID`
  - `PROCMGR_SESSION_SIGNAL_PGRP`
  - `PROCMGR_SESSION_SIGNAL_LEADER`

For more information, see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `procmgr_session()` function provides session support to character device terminal drivers in their resource managers, C library functions, and session management applications.

The arguments that you provide need to match the event:

- **PROCMGR_SESSION_TCSETSID**
  - Used by the `tcsetsid()` function to set the file descriptor, `id`, to be the controlling terminal for the session headed by the session leader, `sid`.

- **PROCMGR_SESSION_SETSID**
  - Used by the `setsid()` function to create a new session with the calling process becoming the session leader. Pass zero for both `sid` and `id` arguments.

- **PROCMGR_SESSION_SETPGRP**
  - Used by a character device resource manager to change the process group upon the request of a client calling the `tcsetpgrp()` function. Set the `sid` argument to the client’s current session and the `id` argument to the new target process group for the client.

- **PROCMGR_SESSION_SIGNAL_PID**,
- **PROCMGR_SESSION_SIGNAL_PGRP**,
- **PROCMGR_SESSION_SIGNAL_LEADER**
  - Used by a character device resource manager to drop a signal of the type specified as the `id` argument (generally a terminal/job control signal) on the appropriate member of the session specified by the `sid` argument.

Returns:

0 Success.

-1 Failure.
procmgr_session()

Classification:
QNX Neutrino

Safety
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

setsid(), tcsetpgrp(), tcsetsid()
**progsname**

The basename of the program being executed

**Synopsis:**

```c
char * __progsname
```

**Description:**

This global variable holds the basename of the program being executed.

---

This variable isn’t defined in any header file. If you want to refer to it, you need to add your own `extern` statement.

**Classification:**

QNX Neutrino

**See also:**

`cmdfd()`, `cmdname()`
protoent
Structure for information from the protocol database

Synopsis:

```c
#include <netdb.h>

struct protoent {
    char * p_name;
    char ** p_aliases;
    int p_proto;
};
```

Description:

The `protoent` structure holds information from the network protocols database, `/etc/protocols`.

The members of this structure are:

- `p_name` The name of the protocol.
- `p_aliases` A zero-terminated list of alternate names for the protocol.
- `p_proto` The protocol number.

Classification:

POSIX 1003.1

See also:

`endprotoent()`, `getprotobynumber()`, `getprotobyname()`, `getprotoent()`, `setprotoent()`

`/etc/protocols` in the `Utilities Reference`
**pthread_abort()**

Unconditionally terminate the target thread

**Synopsis:**

```c
#include <pthread.h>

int pthread_abort( pthread_t thread );
```

**Arguments:**

- `thread` The ID of the thread that you want to terminate, which you can get when you call `pthread_create()` or `pthread_self()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_abort()` function terminates the target thread. Termination takes effect immediately and isn’t a function of the cancelability state of the target thread. No cancellation handlers or thread-specific-data destructor functions are executed. Thread abortion doesn’t release any application-visible process resources, including, but not limited to, mutexes and file descriptors. (The behavior of POSIX calls following a call to `pthread_abort()` is unspecified.)

The status of PTHREAD_ABORTED is available to any thread joining with the target thread. The constant PTHREAD_ABORTED expands to a constant expression, of type `void *`. Its value doesn’t match any pointer to an object in memory, or the values NULL and PTHREAD_CANCELED.

The side effects of aborting a thread that’s suspended during a call of a POSIX 1003.1 function are the same as the side effects that may be seen in a single-threaded process when a call to a POSIX 1003.1 function is interrupted by a signal and the given function returns EINTR. Any such side effects occur before the thread terminates.
Returns:

EOK   Success.
ESRCH No thread with the given ID was found.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cancel(), pthread_detach(), pthread_exit(), ThreadDestroy()
pthread_atfork()
Register fork handlers

Synopsis:
#include <process.h>

int pthread_atfork( void (*prepare)(void),
                    void (*parent)(void),
                    void (*child)(void) );

Arguments:
prepare  NULL, or a pointer to the handler to call before the fork.
parent   NULL, or a pointer to the handler to call after the fork in
          the parent process.
child    NULL, or a pointer to the handler to call after the fork in
          the child process.

Library:
libc
Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:
The pthread_atfork() function registers fork handler functions to be
called before and after a fork(), in the context of the thread that called
fork(). You can set one or more of the arguments to NULL to indicate
no handler.

You can register multiple prepare, parent, and child fork handlers, by
making additional calls to pthread_atfork(). In this case, the parent
and child handlers are called in the order they were registered, and the
prepare handlers are called in the reverse order.
You can’t use the `pthread_atfork()` function for useful purposes as the C library doesn’t have the necessary handlers. It also implies that Neutrino currently doesn’t support `fork()` in multi-threaded programs.

**Returns:**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory to record fork handlers.</td>
</tr>
</tbody>
</table>

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atexit()`, `fork()`
**pthread_attr_destroy()**

Destroy a thread-attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_destroy( pthread_attr_t * attr );
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that you want to destroy.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_destroy()` function destroys the given thread-attribute object.

The QNX implementation of this function doesn’t actually free the memory used by the `pthread_attr_t` structure. To conform to the POSIX standard, don’t reuse the attribute object unless you reinitialize it by calling `pthread_attr_init()`.

You can use a thread-attribute object to define the attributes of new threads when you call `pthread_create()`.

**Returns:**

0 for success, or an error number.

**Errors:**

- **EOK** Success.
Classification:

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_attr_init()`, `pthread_create()`
**pthread_attr_getdetachstate()** © 2007, QNX Software Systems GmbH & Co. KG.

Get thread detach state attribute

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_getdetachstate(
    const pthread_attr_t* attr,
    int* detachstate);
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `detachstate` A pointer to a location where the function can store the thread detach state. For more information, see `pthread_attr_setdetachstate()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_getdetachstate()` function gets the thread detach state attribute from the thread attribute object `attr` and returns it in `detachstate`.

**Returns:**

- `EOK` Success.

**Classification:**

POSIX 1003.1 THR
**pthread_attr_getdetachstate()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_attr_setdetachstate()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_getguardsize()** © 2007, QNX Software Systems GmbH & Co. KG.

Get the size of the thread’s guard area

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_getguardsize(
    const pthread_attr_t* attr,
    size_t* guardsize
);
```

**Arguments:**

- **attr**
  A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.

- **guardsize**
  A pointer to a location where the function can store the size of the thread’s guard area. For more information, see `pthread_attr_setguardsize()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_getguardsize()` function gets the value of the thread guardsize attribute from the attribute structure `attr`.

**Returns:**

- **EOK**
  Success.

- **EINVAL**
  Invalid pointer, `attr`, to a `pthread_attr_t` structure.

**Classification:**

POSIX 1003.1 XSI
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_attr_getguardsize()`
Get a thread’s inherit-scheduling attribute

Synopsis:

```c
#include <pthread.h>

int pthread_attr_getinheritsched(
    const pthread_attr_t* attr,
    int* inheritsched);
```

Arguments:

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.

- `inheritsched` A pointer to a location where the function can store the value of the inherit-scheduling attribute. For more information, see `pthread_attr_setinheritsched()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getinheritsched()` function gets the thread inherit-scheduling attribute from the attribute object `attr` and returns it in `inheritsched`.

The inherit-scheduling attribute determines whether a thread inherits the scheduling policy of its parent.

Returns:

- `EOK` Success.
Classification:

POSIX 1003.1 THR TPS

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_attr_setinheritsched(), pthread_attr_init(), pthread_create()
**Synopsis:**

```c
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedparam(
    const pthread_attr_t * attr,
    struct sched_param * param);
```

**Arguments:**
- **attr** A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- **param** A pointer to a `sched_param` structure where the function can store the current scheduling parameters.

**Library:**
- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_getschedparam()` function gets the thread scheduling parameters attribute from the thread attribute object `attr` and returns it in `param`.

**Returns:**
- **EOK** Success.

**Classification:**
- POSIX 1003.1 THR
### pthread_attr_getschedparam()

#### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### See also:

`pthread_attr_getschedparam()`, `pthread_attr_init()`, `pthread_create()`, `sched_param`
Get thread scheduling policy attribute

Synopsis:

```c
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedpolicy(
    const pthread_attr_t* attr,
    int* policy );
```

Arguments:

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `policy` The current thread scheduling policy. For more information, see `pthread_attr_setschedpolicy()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getschedpolicy()` function gets the thread scheduling policy attribute from the thread attribute object `attr` and returns it in `policy`.

Returns:

- `EOK` Success.

Classification:

POSIX 1003.1 THR TPS
### pthread_attr_getschedpolicy()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_attr_setschedpolicy()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_getscope()**  © 2007, QNX Software Systems GmbH & Co. KG.

*Get thread contention scope attribute*

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_getscope(
    const pthread_attr_t *attr,
    int *scope);
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `scope` A pointer to a location where the function can store the current contention scope. For more information, see `pthread_attr_setscope()`.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_getscope()` function gets the thread contention scope attribute from the thread attribute object `attr` and returns it in `scope`.

**Returns:**

- EOK Success.

**Classification:**

- POSIX 1003.1 THR TPS
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_attr_setscope()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_getstackaddr()** © 2007, QNX Software Systems GmbH & Co. KG.

Get thread stack address attribute

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_getstackaddr(
    const pthread_attr_t* attr,
    void** stackaddr);
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `stackaddr` A pointer to a location where the function can store the address of the thread stack.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_getstackaddr()` function gets the thread stack address attribute from the thread attribute object `attr` and returns it in `stackaddr`.

For more information about the thread stack, see `pthread_attr_setstackaddr()`

**Returns:**

- `EOK` Success.
pthread_attr_getstackaddr()

Classification:

POSIX 1003.1 THR TSA

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

pthread_attr_setstackaddr(), pthread_attr_init(), pthread_create().
**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_getstacklazy(
    const pthread_attr_t * attr,
    int * lazystack
);
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `lazystack` A pointer to a location where the function can store the current lazy-stack attribute. For more information, see `pthread_attr_setstacklazy()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_getstacklazy()` function gets the thread lazy-stack attribute in the attribute object `attr` and stores it in the location pointed to by `lazystack`.

**Returns:**

- `EOK` Success.

**Classification:**

QNX Neutrino
pthread_attr_getstacklazy()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_attr_setinheritsched(), pthread_attr_setstacklazy()
Get the amount of memory to preallocate for a thread’s MAP_LAZY stack

Synopsis:

```c
#include <pthread.h>

int pthread_attr_getstackprealloc(
    const pthread_attr_t * attr,
    size_t * stacksize);
```

Arguments:

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `stacksize` A pointer to a location where the function can store the size of the stack.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getstackprealloc()` function gets the amount of memory to preallocate for a thread’s MAP_LAZY stack.

This function was added in the QNX Neutrino Core OS 6.3.2.

Returns:

- `EOK` Success.

Classification:

QNX Neutrino
pthread_attr_getstackprealloc()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_attr_getstacksize(), pthread_attr_init(),
pthread_attr_setstackprealloc(), pthread_create()
Get the thread stack-size attribute

Synopsis:

```c
#include <pthread.h>

int pthread_attr_getstacksize(
    const pthread_attr_t* attr,
    size_t* stacksize);
```

Arguments:

| attr      | A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`. |
| stacksize | A pointer to a location where the function can store the stack size to be used for new threads. |

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_getstacksize()` function gets the thread stack size attribute from the thread attribute object `attr` and returns it in `stacksize`.

You can set the stack size by calling `pthread_attr_setstacksize()`.

Returns:

- EOK Success.

Classification:

POSIX 1003.1 THR TSS
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_attr_getpagesize(), pthread_attr_init(), pthread_create().
**pthread_attr_init()**

Initialize a thread-attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_init( pthread_attr_t *attr );
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that you want to initialize. For more information, see below.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_init()` function initializes the thread attributes in the thread attribute object `attr` to their default values:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>detachstate</code></td>
<td>PTHREAD_CREATE_JOINABLE</td>
</tr>
<tr>
<td><code>schedpolicy</code></td>
<td>PTHREAD_INHERIT_SCHED</td>
</tr>
<tr>
<td><code>schedparam</code></td>
<td>Inherited from parent thread</td>
</tr>
<tr>
<td><code>contentionscope</code></td>
<td>PTHREAD_SCOPE_SYSTEM</td>
</tr>
<tr>
<td><code>stacksize</code></td>
<td>4K bytes</td>
</tr>
<tr>
<td><code>stackaddr</code></td>
<td>NULL</td>
</tr>
</tbody>
</table>

After initialization, you can use the `pthread_attr_` * family of functions to get and set the attributes:
You can also set some non-POSIX attributes; for more information, see “QNX extensions,” in the documentation for `pthread_create()`.

You can then pass the attribute object to `pthread_create()` to create a thread with the required attributes. You can use the same attribute object in multiple calls to `pthread_create()`.

The effect of initializing an already-initialized thread-attribute object is undefined.

**Returns:**

EOK Success.

**Classification:**

POSIX 1003.1 THR

**Safety**

Cancellation point No

continued...
**pthread_attr_init()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

* pthread_attr_destroy(), pthread_create() *
**pthread_attr_setdetachstate()**

*Set thread detach state attribute*

### Synopsis:

```c
#include <pthread.h>

int pthread_attr_setdetachstate(
    pthread_attr_t* attr,
    int detachstate);
```

### Arguments:

- **attr**
  
  A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.

- **detachstate**

  The new value for the thread detach state:
  
  - PTHREAD_CREATE_JOINABLE — create the thread in a joinable state.
  
  - PTHREAD_CREATE_DETACHED — create the thread in a detached state.

### Library:

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

### Description:

The `pthread_attr_setdetachstate()` function sets the thread detach state attribute in the thread attribute object `attr` to `detachstate`.

The default value for the thread detach state is `PTHREAD_CREATE_JOINABLE`. 

---

September 10, 2007

C Library — P to R 2153
**pthread_attr_setdetachstate()** © 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

- EOK  Success.
- EINVAL  Invalid thread detach state value.

**Classification:**

POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_attr_getdetachstate()`, `pthread_attr_init()`, `pthread_create()`, `pthread_detach()`, `pthread_join()`.

2154  C Library — P to R  September 10, 2007
Synopsis:

```c
#include <pthread.h>

int pthread_attr_setguardsize(
    pthread_attr_t* attr,
    size_t guardsize );
```

Arguments:

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `guardsize` The new value for the size of the thread’s guard area.

Library:

- `libc`
  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setguardsize()` function sets the size of the thread’s guard area in the attribute structure `attr` to `guardsize`.

If `guardsize` is 0, threads created with `attr` have no guard area; otherwise, a guard area of at least `guardsize` bytes is provided. You can get the default `guardsize` value by specifying `_SC_PAGESIZE` in a call to `sysconf()`.

The guardsize attribute controls the size of the guard area for the thread’s stack. This guard area helps protect against stack overflows; `guardsize` bytes of extra memory is allocated at the overflow end of the stack. If a thread overflows into this buffer, it receives a SIGSEGV signal.

The guardsize attribute is provided because:

- Stack overflow protection can waste system resources. An application that creates many threads can save system resources by...
thread_attr_setguardsize() © 2007, QNX Software Systems GmbH & Co. KG.

turning off guard areas if it trusts its threads not to overflow the stack.
- When threads allocate large objects on the stack, a large guardsize is required to detect stack overflows.

Returns:

- EOK Success.
- EINVAL Invalid pointer, attr, to a pthread_attr_t structure, or guardsize is invalid.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

If you provide a stack (using attr's stackaddr attribute; see pthread_attr_setstackaddr()), the guardsize is ignored, and there's no stack overflow protection for that thread.

The guardsize argument is completely ignored when using a physical mode memory manager.

See also:

- pthread_attr_getguardsize(), pthread_attr_init(),
- pthread_attr_setstackaddr(), sysconf()
Synopsis:

#include <pthread.h>

int pthread_attr_setinheritsched(
    pthread_attr_t * attr,
    int inheritsched
);

Arguments:

attr
A pointer to the pthread_attr_t structure that defines the attributes to use when creating new threads. For more information, see pthread_attr_init().

inheritsched
The new value for the thread’s inherit-scheduling attribute:

- PTHREAD_INHERIT_SCHED — the thread inherits the scheduling policy of the parent thread.
- PTHREAD_EXPLICIT_SCHED — use the scheduling policy specified in attr for the thread.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The pthread_attr_setinheritsched() function sets the thread inherit scheduling attribute in the attribute object attr to inheritsched.

The default value of the thread inherit scheduling attribute is PTHREAD_INHERIT_SCHED.
Returns:

- **EOK**: Success.
- **EINVAL**: Invalid thread attribute object `attr`.
- **ENOTSUP**: Invalid thread inherit scheduling attribute `inheritsched`.

Classification:

- POSIX 1003.1 THR TPS

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `pthread_attr_getinheritsched()`,
- `pthread_attr_init()`,
- `pthread_create()`.
**pthread_attr_setschedparam()**

Set a thread’s scheduling parameters attribute

**Synopsis:**

```c
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(
    pthread_attr_t * attr,
    const struct sched_param * param
);
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `param` A pointer to a `sched_param` structure that defines the thread’s scheduling parameters.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_setschedparam()` function sets the thread scheduling parameters attribute in the thread attribute object `attr` to `param`.

The thread scheduling parameters are used only if you’ve set the thread inherit scheduling attribute to `PTHREAD_EXPLICIT_SCHED` by calling `pthread_attr_setinheritsched()`. By default, a thread inherits its parent’s priority.

**Returns:**

- **EOK** Success.
- **EINV AL** Invalid thread attribute object `attr`.
- **ENOTSUP** Invalid thread scheduling parameters attribute `param`.
**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_attr_setschedparam()`, `pthread_attr_setinheritsched()`, `pthread_attr_init()`, `pthread_create()`, `sched_param`
Synopsis:

```c
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedpolicy(
    pthread_attr_t* attr,
    int policy );
```

Arguments:

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `policy` The new value for the scheduling policy:
  - `SCHED_FIFO` — first-in first-out scheduling.
  - `SCHED_RR` — round-robin scheduling.
  - `SCHED_OTHER` — currently the same as `SCHED_RR`.
  - `SCHED_NOCHANGE` — don’t change the policy.
  - `SCHED_SPORADIC` — sporadic scheduling.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_attr_setschedpolicy()` function sets the thread scheduling policy attribute in the thread attribute object `attr` to `policy`.

The `policy` attribute is used only if you’ve set the thread inherit-scheduling attribute to `PTHREAD_EXPLICIT_SCHED` by calling `pthread_attr_setinheritsched()`.

For descriptions of the scheduling policies, see “Scheduling algorithms” in the chapter on the Neutrino microkernel in the System Architecture guide.
Returns:

- **EOK**: Success.
- **EINVAL**: Invalid thread attribute object `attr`.
- **ENOTSUP**: Invalid thread scheduling policy `policy`.

Classification:

- **POSIX 1003.1 THR TPS**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `pthread_attr_getschedpolicy()`, `pthread_attr_init()`, `pthread_create()`.
**pthread_attr_setscope()**

*Set thread contention scope attribute*

### Synopsis:

```c
#include <pthread.h>

int pthread_attr_setscope( pthread_attr_t* attr,
                          int scope );
```

### Arguments:

- **attr**  
  A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.

- **scope**  
  The new value for the contention scope attribute:
  - `PTHREAD_SCOPE_SYSTEM` — schedule all threads together.

### Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `pthread_attr_setscope()` sets the thread contention scope attribute in the thread attribute object `attr` to `scope`.

### Returns:

- **EOK**  
  Success.

- **EINVAL**  
  Invalid thread attribute object `attr`.

- **ENOTSUP**  
  Invalid thread contention scope attribute `scope`. 
Classification:

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_attr_getscope()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_setstackaddr()**

Set the thread stack address attribute

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_setstackaddr( pthread_attr_t * attr, 
                                void * stackaddr );
```

**Arguments:**

- `attr`: A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `stackaddr`: A pointer to the block of memory that you want a new thread to use as its stack.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_setstackaddr()` function sets the thread stack address attribute in the attribute object `attr` to `stackaddr`.

The default value for the thread stack address attribute is NULL. A thread created with a NULL stack address attribute will have a stack dynamically allocated by the system of minimum size `PTHREAD_STACK_MIN`. If the system allocates a stack, it reclaims the space when the thread terminates. If you allocate a stack, you must free it.

**Returns:**

- EOK: Success.
Classification:

POSIX 1003.1 THR TSA

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The QNX interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```c
void nothingthread( void )
{
    return;
}
```

See also:

`pthread_attr_setstackaddr()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_setstacklazy()**

Set the thread lazy-stack attribute

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_setstacklazy(
    pthread_attr_t * attr,
    int lazystack );
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.

- `lazystack` One of:
  - `PTHREAD_STACK_LAZY` — allocate physical memory for the thread stack on demand (the default).
  - `PTHREAD_STACK_NOTLAZY` — allocate physical memory for the whole stack up front. Use this value to ensure that your server processes don’t die later on because they’re unable to allocate stack memory. We recommend that you set the stack size as well, because the default stack size is probably much larger than you really need.

**Library:**

`libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_setstacklazy()` function sets the thread stack attribute in the attribute object `attr` to `lazystack`. 
Returns:

- **EOK** Success.
- **EINV AL** The thread-attribute object that `attr` points to is invalid.
- **ENOTSUP** The value of `lazystack` is invalid.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`pthread_attr_getstacklazy()`, `pthread_attr_setinheritsched()`
pthread_attr_setstackprealloc()
Set the amount of memory to preallocate for a thread's MAP_LAZY stack

Synopsis:
#include <pthread.h>

int pthread_attr_setstackprealloc(
    const pthread_attr_t * attr,
    size_t stacksize);

Arguments:

attr
A pointer to the pthread_attr_t structure that defines the attributes to use when creating new threads. For more information, see pthread_attr_init().

stacksize
The amount of stack you want to preallocate for new threads.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The pthread_attr_setstackprealloc() function sets the size of the memory to preallocate for a thread’s MAP_LAZY stack. By default, the system allocates sysconf(_SC_PAGESIZE) bytes of physical memory for the initial stack reference. This function allows you to change this default memory size, if you know that a thread will need more stack space. Semantically, there is no difference in operation, but the memory manager attempts to make more efficient use of Memory Management Unit hardware (e.g. a larger page size in the page entry table) for the stack if it knows upfront that more memory will be required.
This function was added in the QNX Neutrino Core OS 6.3.2.

Returns:

EOK  Success.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_attr_setstackprealloc()`, `pthread_attr_getstacksize()`, `pthread_attr_init()`, `pthread_create()`
**pthread_attr_setstacksize()**

Set the thread stack-size attribute

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_setstacksize( pthread_attr_t * attr, 
                                size_t stacksize );
```

**Arguments:**

- `attr` A pointer to the `pthread_attr_t` structure that defines the attributes to use when creating new threads. For more information, see `pthread_attr_init()`.
- `stacksize` The size of the stack you want to use in new threads. The minimum value of the thread stack-size attribute is PTHREAD_STACK_MIN.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_attr_setstacksize()` function sets the thread stack size attribute in the thread attribute object `attr` to `stacksize`.

**Returns:**

- `EOK` Success.
- `EINVAL` The value of `stacksize` is less than PTHREAD_STACK_MIN or greater than the system limit.
Classification:

POSIX 1003.1 THR TSS

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The QNX interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```c
void nothingthread( void )
{
    return;
}
```

See also:

`pthread_attr_getstacksize()`, `pthread_attr_init()`, `pthread_create()`. 


**pthread_barrier_destroy()**

*Destroy a barrier object*

**Synopsis:**

```c
#include <pthread.h>

int pthread_barrier_destroy(
    pthread_barrier_t * barrier )
```

**Arguments:**

- `barrier`: A pointer to the `pthread_barrier_t` object that you want to destroy.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_barrier_destroy()` function destroys the barrier referenced by `barrier` and releases any resources used by the barrier. Subsequent use of the barrier is undefined until you reinitialize the barrier by calling `pthread_barrier_init()`.

**Returns:**

- `EBUSY`: The `barrier` is in use.
- `EINVAL`: Invalid `barrier`.
- `EOK`: Success.

**Classification:**

POSIX 1003.1 THR BAR
**pthread_barrier_destroy()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_barrier_init(), pthread_barrier_wait()`
**pthread_barrier_init()**

*Initialize a barrier object*

**Synopsis:**

```c
#include <pthread.h>

int pthread_barrier_init(
    pthread_barrier_t * barrier,
    const pthread_barrierattr_t * attr
    unsigned int count );
```

**Arguments:**

- **barrier**
  A pointer to the `pthread_barrier_t` object that you want to initialize.

- **attr**
  NULL, or a pointer to a `pthread_barrierattr_t` structure that specifies the attributes that you want to use for the barrier.

- **count**
  The number of threads that must call `pthread_barrier_wait()` before any of them successfully returns from the call. This value must be greater than zero.

**Library:**

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `pthread_barrier_init()` function allocates any resources required to use the barrier referenced by `barrier` and initializes the barrier with attributes referenced by `attr`. If `attr` is NULL, the default barrier attributes are used. The effect is the same as passing the address of a default barrier attributes object. Once it’s initialized, you can use the barrier any number of times without reinitializing it.

If `pthread_barrier_init()` fails, the barrier isn’t initialized.
In cases where the default barrier attributes are appropriate, you can use `PTHREAD_BARRIER_INITIALIZER()` macro to initialize barriers that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_barrier_init()` with parameter `attr` specified as NULL, except that no error checks are performed.

**Returns:**

- **EAGAIN** The system lacks the necessary resources to initialize another barrier.
- **EBUSY** Attempt to reinitialize a barrier while it’s in use.
- **EFAULT** A fault occurred when the kernel tried to access `barrier` or `attr`.
- **EINVAL** Invalid value specified by `attr`.
- **EOK** Success.

**Classification:**

POSIX 1003.1 THR BAR

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_barrierattr_init()`, `pthread_barrier_destroy()`, `pthread_barrier_wait()`
**Synopsis:**

```c
#include <sync.h>

int pthread_barrier_wait( pthread_barrier_t * barrier );
```

**Arguments:**

- `barrier` A pointer to the `pthread_barrier_t` object that you want to use to synchronize the threads. You must initialize the barrier by calling `pthread_barrier_init()`, before calling `pthread_barrier_wait()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_barrier_wait()` function synchronizes participating threads at the barrier referenced by `barrier`. The calling thread blocks — that is, doesn’t return from `pthread_barrier_wait()` — until the required number of threads have called `pthread_barrier_wait()`, specifying the barrier.

When the required number of threads have called `pthread_barrier_wait()` specifying the barrier, the constant `BARRIER_SERIAL_THREAD` is returned to one unspecified thread, and zero is returned to each of the remaining threads. At this point, the barrier is reset to the state it occupied as a result of the most recent `pthread_barrier_init()` function that referenced it.

The constant `BARRIER_SERIAL_THREAD` is defined in `<pthread.h>`, and its value is distinct from any other value that `pthread_barrier_wait()` returns.

If a signal is delivered to a thread blocked on a barrier, on return from the signal handler, the thread resumes waiting at the barrier as if it hadn’t been interrupted.
**pthread_barrier_wait()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the barrier and zero for each of the other threads; otherwise, an error number is returned:

EINVAL The barrier argument isn’t initialized.

**Classification:**

POSIX 1003.1 THR BAR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

pthread_barrier_destroy(), pthread_barrier_init()
pthread_barrierattr_destroy()

Destroy a barrier-attributes object

Synopsis:

```c
#include <pthread.h>

int pthread_barrierattr_destroy(
    pthread_barrierattr_t * attr);
```

Arguments:

`attr` A pointer to the `pthread_barrierattr_t` object that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_barrierattr_destroy()` function destroys the barrier-attributes object `attr`. Subsequent use of the object is undefined until you reinitialize the object by calling `pthread_barrierattr_init()`.

Once you’ve used a barrier-attributes object to initialize one or more barriers, any changes to the attributes object (including destroying it) don’t affect any previously initialized barriers.

Returns:

EOK Success.

EINVAL Invalid barrier attribute object `attr`.

Classification:

POSIX 1003.1 THR BAR
```markdown
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_barrier_destroy()`, `pthread_barrierattr_getpshared()`, `pthread_barrierattr_init()`, `pthread_barrierattr_setpshared()`
Get the value of a barrier’s process-shared attribute

Synopsis:

```c
#include <pthread.h>

int pthread_barrierattr_getpshared(
    const pthread_barrierattr_t * attr
    int * pshared);
```

Arguments:

- **attr**: A pointer to the `pthread_barrierattr_t` object whose attribute you want to query. You must have initialized this object by calling `pthread_barrierattr_init()` before calling `pthread_barrierattr_getpshared()`.

- **pshared**: A pointer to a location where the function can store the value of the process-shared attribute. For information about the possible values, see `pthread_barrierattr_setpshared()`.

Library:

- libc

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_barrierattr_getpshared()` function gets the value of the process-shared attribute from the attributes object referenced by `attr` and stores the value in the object referenced by `pshared`.

Returns:

- EOK: Success.
- EINVAL: Invalid barrier attribute object `attr`.
Classification:

POSIX 1003.1 THR BAR TSH

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

 pthread_barrier_init(), pthread_barrierattr_destroy(),
 pthread_barrierattr_init(), pthread_barrierattr_setpshared()
**Synopsis:**
```
#include <pthread.h>

int pthread_barrierattr_init(
    pthread_barrierattr_t * attr);
```

**Arguments:**
- `attr` A pointer to the `pthread_barrierattr_t` object that you want to initialize.

**Library:**
- libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**
The `pthread_barrierattr_init()` function initializes the barrier attributes object `attr` with the default value for all of the attributes.

**Returns:**
- EOK Success.
- ENOMEM Insufficient memory to initialize barrier attribute object `attr`.

**Classification:**
- POSIX 1003.1 THR BAR

**Safety**
- Cancellation point: No
- Interrupt handler: No

continued...
**pthread_barrierattr_init()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_barrier_init()`, `pthread_barrierattr_destroy()`,
`pthread_barrierattr_getpshared()`, `pthread_barrierattr_setpshared()`
**Synopsis:**

```c
#include <sync.h>

int pthread_barrierattr_setpshared(
    pthread_barrierattr_t * attr
    int pshared );
```

**Arguments:**

- **attr** A pointer to the `pthread_barrierattr_t` object whose attribute you want to set. You must have initialized this object by calling `pthread_barrierattr_init()` before calling `pthread_barrierattr_setpshared()`.
- **pshared** The new value of the process-shared attribute; one of:
  - `PTHREAD_PROCESS_SHARED` — allow a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated.
  - `PTHREAD_PROCESS_PRIVATE` (default behavior) — allow a barrier to be operated on only by threads created within the same process as the thread that initialized the barrier. If threads of different processes attempt to operate on such a barrier, the behavior is as if `PTHREAD_PROCESS_SHARED` were set.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_barrierattr_setpshared()` function sets the process-shared attribute in an initialized attributes object referenced by `attr`. 
Returns:

EOK Success.
EINV AL The attribute object, attr, or the new value specified in pshared isn’t valid.

Classification:

POSIX 1003.1 THR BAR TSH

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point       No</td>
</tr>
<tr>
<td>Interrupt handler        No</td>
</tr>
<tr>
<td>Signal handler           Yes</td>
</tr>
<tr>
<td>Thread                  Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_barrier_init()  pthread_barrierattr_destroy(),
pthread_barrierattr_getpshared(), pthread_barrierattr_init()
pthread_cancel()

Cancel a thread

Synopsis:

```
#include <pthread.h>

int pthread_cancel( pthread_t thread );
```

Arguments:

- `thread` The ID of the thread that you want to cancel, which you can get when you call `pthread_create()` or `pthread_self()`.

Library:

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `pthread_cancel()` function requests that the target thread `thread` be canceled (terminated). The cancellation type and state of the target determine when the cancellation takes effect.

When the cancellation is acted on, the target’s cancellation cleanup handlers are called. When the last cancellation cleanup handler returns, the target’s thread-specific-data destructor functions are called. When the last destructor function returns, the target is terminated. Cancellation processing in the target thread runs asynchronously with respect to the calling thread.

Returns:

- `EOK` Success.
- `ESRCH` No thread with thread ID `thread` exists.
**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_cleanup_push()`, `pthread_cleanup_pop()`,
`pthread_cond_wait()`, `pthread_cond_timedwait()`, `pthread_exit()`,
`pthread_join()`, `pthread_key_create()`, `pthread_setcancelstate()`,
`pthread_setcanceltype()`, `pthread_testcancel()`, `ThreadCancel()`.
**pthread cleanup pop()**

Pop a function off of a thread’s cancellation-cleanup stack

**Synopsis:**

```c
#include <pthread.h>

void pthread_cleanup_pop( int execute );
```

**Arguments:**

`execute`  
Zero if you don’t want to execute the handler; nonzero if you do.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cleanup_pop()` macro pops the top cancellation-cleanup handler from the calling thread’s cancellation-cleanup stack and invokes the handler if `execute` is nonzero.

The `pthread_cleanup_pop()` macro expands to a few lines of code that end with a closing brace (`}`), but don’t have a matching opening brace (`{`). You must pair `pthread_cleanup_pop()` with `pthread_cleanup_push()` within the same lexical scope.

**Examples:**

See `pthread_cleanup_push()`.

**Classification:**

POSIX 1003.1 THR
**pthread_cleanup_pop()**

---

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_cleanup_push()`, `pthread_cancel()`, `pthread_exit()`
**pthread_cleanup_push()**

*Push a function onto a thread’s cancellation-cleanup stack*

**Synopsis:**

```c
#include <pthread.h>

void pthread_cleanup_push( void (routine)(void*),
                           void* arg );
```

**Arguments:**

- `routine` The handler that you want to push onto the thread’s stack.
- `arg` A pointer to whatever data you want to pass to the function when it’s called.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cleanup_push()` function pushes the given cancellation-cleanup handler `routine` onto the calling thread’s cancellation-cleanup stack.

The cancellation-cleanup handler is popped from the stack and invoked with argument `arg` when the thread:

- exits (i.e. calls `pthread_exit()`)
- acts on a cancellation request
- calls `pthread_cleanup_pop()` with a nonzero argument.
**pthread_cleanup_push()** © 2007, QNX Software Systems GmbH & Co. KG.

The `pthread_cleanup_push()` macro expands to a few lines of code that start with an opening brace (`{`), but don’t have a matching closing brace (`}`). You must pair `pthread_cleanup_push()` with `pthread_cleanup_pop()` within the same lexical scope.

**Examples:**

Use a cancellation cleanup handler to free resources, such as a mutex, when a thread is terminated:

```c
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void unlock( void * arg )
{
    pthread_mutex_unlock( &lock );
}

void * function( void * arg )
{
    while( 1 )
    {
        pthread_mutex_lock( &lock );
        pthread_cleanup_push( &unlock, NULL );

        /*
         * Any of the possible cancellation points could go here.
         */
        pthread_testcancel();

        pthread_cleanup_pop( 1 );
    }
}
```

**Classification:**

POSIX 1003.1 THR
## pthread_cleanup_push()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

(pthread_cleanup_pop(), pthread_cancel(), pthread_exit())
**pthread_cond_broadcast()** © 2007, QNX Software Systems GmbH & Co. KG.

*Unblock threads waiting on condition*

**Synopsis:**
```
#include <pthread.h>

int pthread_cond_broadcast( pthread_cond_t* cond );
```

**Arguments:**
- `cond` A pointer to the `pthread_cond_t` object for which you want to unblock the threads.

**Library:**
- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `pthread_cond_broadcast()` function unblocks all threads currently blocked on the condition variable `cond`. The threads are unblocked in priority order.

If more than one thread at a particular priority is blocked, those threads are unblocked in FIFO order.

**Returns:**
- **EOK** Success.
- **EFAULT** A fault occurred trying to access the buffers provided.
- **EINVAL** Invalid condition variable `cond`.

**Classification:**
- POSIX 1003.1 THR
**pthread_cond_broadcast()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_cond_signal()`, `pthread_cond_wait()`, `SyncCondvarSignal()`
**pthread_cond_destroy()**

*Destroy condition variable*

**Synopsis:**

```c
#include <pthread.h>

int pthread_cond_destroy( pthread_cond_t* cond );
```

**Arguments:**

- `cond` A pointer to the `pthread_cond_t` object that you want to destroy.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cond_destroy()` function destroys the condition variable `cond`. After you’ve destroyed a condition variable, you shouldn’t reuse it until you’ve reinitialized it by calling `pthread_cond_init()`.

**Returns:**

- `EOK` Success.
- `EBUSY` Another thread is blocked on the condition variable `cond`.
- `EINVAL` Invalid condition variable `cond`.

**Classification:**

POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
</table>

continued…
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_init(), SyncDestroy()
**pthread_cond_init()**

Initialize a condition variable

**Synopsis:**

```c
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init( pthread_cond_t* cond, 
                        pthread_condattr_t* attr );
```

**Arguments:**

- `cond`: A pointer to the `pthread_cond_t` object that you want to initialize.
- `attr`: NULL, or a pointer to a `pthread_condattr_t` object that specifies the attributes that you want to use for the condvar. For more information, see `pthread_condattr_init()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cond_init()` function initializes the condition variable `cond` with the attributes in the condition variable attribute object `attr`. If `attr` is NULL, `cond` is initialized with the default values for the attributes.

If the condition variable is statically allocated, you can initialize it with the default attribute values by assigning to it the macro `PTHREAD_COND_INITIALIZER`.

Condition variables have at least the following attributes defined:

- `PTHREAD_PROCESS_PRIVATE`

  The condition variable can only be accessed by threads created within the same process as the thread that initialized the condition variable.
**pthread_cond_init()**

PTHREAD_PROCESS_SHARED

Any thread that has access to the memory where the condition variable is allocated can access the condition variable. For more information about these attributes, see

`pthread_condattr_getpshared()` and `pthread_condattr_setpshared()`.

**Returns:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>All kernel synchronization objects are in use.</td>
</tr>
<tr>
<td>EBUSY</td>
<td>Previously initialized condition variable, <code>cond</code>, hasn’t been destroyed.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access <code>cond</code> or <code>attr</code>.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value specified by <code>cond</code> is invalid.</td>
</tr>
</tbody>
</table>

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_condattr_init()`, `pthread_cond_destroy()`
**pthread_cond_signal()**

Unblock a thread that's waiting on a condition variable

**Synopsis:**

```c
#include <pthread.h>

int pthread_cond_signal( pthread_cond_t* cond );
```

**Arguments:**

- `cond`: A pointer to the `pthread_cond_t` object for which you want to unblock the highest-priority thread.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cond_signal()` function unblocks the highest priority thread that's waiting on the condition variable, `cond`.

If more than one thread at the highest priority is waiting, `pthread_cond_signal()` unblocks the one that has been waiting the longest.

**Returns:**

- **EOK**: Success.
- **EFAULT**: A fault occurred trying to access the buffers provided.
- **EINVAL**: Invalid condition variable `cond`.

**Classification:**

POSIX 1003.1 THR
### pthread_cond_signal()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

### See also:

- `pthread_cond_broadcast()`, `pthread_cond_wait()`, `SyncCondvarSignal()`
**Synopsis:**

```c
#include <pthread.h>
#include <time.h>

int pthread_cond_timedwait(
    pthread_cond_t* cond,
    pthread_mutex_t* mutex,
    const struct timespec* abstime);
```

**Arguments:**

- `cond`: The condition variable on which to block the thread.
- `mutex`: The mutex associated with the condition variable.
- `abstime`: A pointer to a `timespec` structure that specifies the maximum time to block the thread, expressed as an absolute time.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cond_timedwait()` function blocks the calling thread on the condition variable `cond`, and unlocks the associated mutex `mutex`. The calling thread must have locked `mutex` before waiting on the condition variable. Upon return from the function, the mutex is again locked and owned by the calling thread.

The calling thread is blocked until either another thread performs a signal or broadcast on the condition variable, the absolute time specified by `abstime` has passed, a signal is delivered to the thread, or the thread is canceled (waiting on a condition variable is a cancellation point). In all cases, the thread reacquires the mutex before being unblocked.
Don’t use a recursive mutex with condition variables.

If a thread that’s blocked on a condition variable is canceled, the thread reacquires the mutex that’s guarding the condition variable, so that the thread’s cleanup handlers run in the same state as the critical code before and after the call to this function. If some other thread owns the lock, the canceled thread blocks until the mutex is available.

Make sure that the thread’s cleanup handlers unlock the mutex.

Returns:

EOK Success, or the call was interrupted by a signal.
EAGAIN Insufficient system resources are available to wait on the condition.
EFAULT A fault occurred trying to access the buffers provided.
EINVAL One or more of the following is true:
  • One or more of cond, mutex and abstime is invalid.
  • Concurrent waits or timed waits on cond used different mutexes.
  • The current thread doesn’t own mutex.
ETIMEDOUT The time specified by abstime has passed.

Examples:

Wait five seconds while trying to acquire control over a condition variable:

```c
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
```
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

int main(int argc, char* argv[])
{
    struct timespec to;
    int retval;

    fprintf(stderr, "starting...
*"
    Here’s the interesting bit; we’ll wait for
    five seconds FROM NOW when we call
    pthread_cond_timedwait().
    */
    memset(&to, 0, sizeof to);
    to.tv_sec = time(0) + 5;
    to.tv_nsec = 0;

    if (retval = pthread_mutex_lock(&m)) {
        fprintf(stderr, "pthread_mutex_lock %s
", strerror(retval));
        exit(EXIT_FAILURE);
    }

    if (retval = pthread_cond_timedwait(&c, &m, &to)) {
        fprintf(stderr, "pthread_cond_timedwait %s
", strerror(retval));
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
pthread_cond_timedwait()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_broadcast(), pthread_cond_init(),
pthread_cond_signal(), pthread_cond_wait(), SyncCondvarWait(),
TimerTimeout(), timespec
**pthread_cond_wait()**

*Wait on condition variable*

**Synopsis:**

```c
#include <pthread.h>

int pthread_cond_wait( pthread_cond_t* cond,
                        pthread_mutex_t* mutex );
```

**Arguments:**

- `cond` A pointer to the `pthread_cond_t` object that you want the threads to block on.
- `mutex` The mutex that you want to unlock.

**Library:**

- `libc`

Use the `-l c` option to *qcc* to link against this library. This library is usually included automatically.

**Description:**

The `pthread_cond_wait()` function blocks the calling thread on the condition variable `cond`, and unlocks the associated mutex `mutex`. The calling thread must have locked `mutex` before waiting on the condition variable. On return from the function, the mutex is again locked and owned by the calling thread.

The calling thread is blocked until either another thread performs a signal or broadcast on the condition variable, a signal is delivered to the thread, or the thread is canceled (waiting on a condition variable is a cancellation point). In all cases, the thread reacquires the mutex before being unblocked.

Don’t use a recursive mutex with condition variables.

If a thread that’s blocked on a condition variable is canceled, the thread reacquires the mutex that’s guarding the condition variable, so that the thread’s cleanup handlers run in the same state as the critical

© 2007, QNX Software Systems GmbH & Co. KG.
code before and after the call to this function. If some other thread owns the lock, the canceled thread blocks until the mutex is available.

Make sure that the thread’s cleanup handlers unlock the mutex.

Returns:

- **EOK** Success, or the call was interrupted by a signal.
- **EAGAIN** Insufficient system resources are available to wait on the condition.
- **EFAULT** A fault occurred trying to access the buffers provided.
- **EINVAL** One or more of the following is true:
  - One or more of `cond` or `mutex` is invalid.
  - Concurrent waits on `cond` used different mutexes.
  - The current thread doesn’t own `mutex`.

Examples:

Use condition variables to synchronize producer and consumer threads:

```c
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int condition = 0;
int count = 0;

int consume( void )
{
    while( 1 )
    {
        pthread_mutex_lock( &mutex );
        while( condition == 0 )
            pthread_cond_wait( &cond, &mutex );
        printf( "Consumed %d\n", count );
        condition = 0;
```
**pthread_cond_wait()**

```c
void* produce( void * arg )
{
    while( 1 )
    {
        pthread_mutex_lock( &mutex );
        while( condition == 1 )
            pthread_cond_wait( &cond, &mutex );
        printf( "Produced %d\n", count++ );
        condition = 1;
        pthread_cond_signal( &cond );
        pthread_mutex_unlock( &mutex );
    }
    return( 0 );
}

int main( void )
{
    pthread_create( NULL, NULL, &produce, NULL );
    return consume();
}
```

Classification:

POSIX 1003.1 THR

**Safety**

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

(pthread_cond_broadcast(), pthread_cond_init(),
 pthread_cond_signal(), pthread_cond_timedwait(),
 SyncCondvarWait())
**pthread_condattr_destroy()** © 2007, QNX Software Systems GmbH & Co. KG.

Destroy a condition-variable attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_condattr_destroy(
    pthread_condattr_t* attr);
```

**Arguments:**

- `attr` A pointer to the `pthread_condattr_t` object that you want to destroy.

**Library:**

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_condattr_destroy()` function destroys the condition variable attribute object `attr`. Once you’ve destroyed the condition-variable attribute object, don’t reuse it until you’ve reinitialized it by calling `pthread_condattr_init()`.

**Returns:**

- EOK Success.
- EINVAL Invalid condition variable attribute object `attr`.

**Classification:**

- POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
</table>

continued...
**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

` pthread_condattr_init()`, ` pthread_cond_init()`
Get the clock attribute from a condition-variable attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_condattr_getclock(
    const pthread_condattr_t * attr,
    clockid_t * id );
```

**Arguments:**

- `attr` A pointer to the `pthread_condattr_t` object from which you want to get the clock.
- `id` A pointer to a `clockid_t` object where the function can store the clock ID.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_condattr_getclock()` function obtains the value of the clock attribute from the attributes object referenced by `attr`.

The clock attribute is the clock ID of the clock that’s used to measure the timeout service of `pthread_cond_timedwait()`. The default value of the clock attribute refers to the system clock.

**Returns:**

- `EOK` Success.
- `EINVAL` Invalid value `attr`.
Classification:

POSIX 1003.1 THR CS

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`pthread_cond_init()`, `pthread_cond_timedwait()`, ` pthread_condattr_destroy()`, ` pthread_condattr_getpshared()`, ` pthread_condattr_init()`, ` pthread_condattr_setclock()`, ` pthread_condattr_setpshared()`, ` pthread_create()`
Get the process-shared attribute from a condition variable attribute object

**Synopsis:**
```
#include <pthread.h>

int pthread_condattr_getpshared(
    const pthread_condattr_t* attr,
    int* pshared);
```

**Arguments:**
- `attr` A pointer to the `pthread_condattr_t` object from which you want to get the attribute.
- `pshared` A pointer to a location where the function can store the process-shared attribute. For the possible values, see `pthread_condattr_setpshared()`.

**Library:**
`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `pthread_condattr_getpshared()` function stores, in the memory pointed to by `pshared`, the process-shared attribute from a condition variable attribute object, `attr`.

**Returns:**
- `EOK` Success.
- `EINVAL` Invalid condition variable attribute object specified by `attr`. 
pthread_condattr_getpshared()

Classification:

POSIX 1003.1 THR TSH

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_condattr_init(), pthread_condattr_setpshared(),
pthread_create(), pthread_mutex_init(), pthread_cond_init().
**pthread_condattr_init()**

Initialize the attribute object for creating condition variables

**Synopsis:**

```c
#include <pthread.h>

int pthread_condattr_init( pthread_condattr_t* attr );
```

**Arguments:**

- `attr` A pointer to the `pthread_condattr_t` object that you want to initialize.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_condattr_init()` function initializes the attributes in the condition variable attribute object `attr` to default values. Pass `attr` to `pthread_cond_init()` to define the attributes of the condition variable.

Condition variables have at least the following attributes defined:

- **PTHREAD_PROCESS_PRIVATE**
  
  The condition variable can be accessed only by threads created within the same process as the thread that initialized the condition variable.

- **PTHREAD_PROCESS_SHARED**

  Any thread that has access to the memory where the condition variable is allocated can access the condition variable.

For more information about these attributes, see `pthread_condattr_getpshared()` and `pthread_condattr_setpshared()`.
Returns:

EOK        Success.
ENOMEM     Insufficient memory to initialize the condition variable attribute object \textit{attr}.

Classification:

POSIX 1003.1 THR

\begin{center}
\begin{tabular}{l|c}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}
\end{center}

See also:

\textit{pthread\_condattr\_destroy()}, \textit{pthread\_cond\_init()}
**pthread_condattr_setclock()** © 2007, QNX Software Systems GmbH & Co. KG.

Set the clock attribute of a condition-variable attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_condattr_setclock(
    pthread_condattr_t * attr,
    clockid_t id);
```

**Arguments:**

- `attr` A pointer to the `pthread_condattr_t` object for which you want to set the clock. You must have initialized this object by calling `pthread_condattr_init()` before calling `pthread_condattr_setclock()`.
- `id` A `clockid_t` object that specifies the ID of the clock that you want to use for the condition variable.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_condattr_setclock()` function sets the clock attribute in an initialized attributes object referenced by `attr`. If `pthread_condattr_setclock()` is called with an `id` argument that refers to a CPU-time clock, the call fails.

The clock attribute is the clock ID of the clock that you want to use to measure the timeout service of `pthread_cond_timedwait()`. The default value of the clock attribute refers to the system clock.

**Returns:**

- `EOK` Success.
- `EINVAL` Invalid value `attr`.

© 2007, QNX Software Systems GmbH & Co. KG.
**pthread_condattr_setclock()**

**Classification:**

POSIX 1003.1 THR CS

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

pthread_cond_init(), pthread_cond_timedwait(),
pthread_condattr_destroy(), pthread_condattr_getclock(),
pthread_condattr_getpshared(), pthread_condattr_init(),
pthread_condattr_setpshared(), pthread_create()
Set the process-shared attribute in a condition variable attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_condattr_setpshared(
    pthread_condattr_t* attr,
    int pshared
);
```

**Arguments:**

- `attr` A pointer to the `pthread_condattr_t` object for which you want to set the attribute.
- `pshared` The new value of the process-shared attribute; one of:
  - `PTHREAD_PROCESS_SHARED` — any thread that has access to the memory where the condition variable is allocated can operate on it, even if the condition variable is allocated in memory that’s shared by multiple processes.
  - `PTHREAD_PROCESS_PRIVATE` — the condition variable can be accessed only by threads created within the same process as the thread that initialized the condition variable; if threads from other processes try to access the `PTHREAD_PROCESS_PRIVATE` condition variable, the behavior is undefined.

**Library:**

`libc`

Use the `-l c` option to `gcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_condattr_setpshared()` function sets the process-shared attribute in a condition variable attribute object, `attr` to the value given by `pshared`.

The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`. 
Returns:

- **EOK**  Success.
- **EINVAL**  The condition variable attribute object specified by `attr`, or the new value specified in `pshared` isn’t valid.

Classification:

POSIX 1003.1 THR TSH

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `pthread_condattr_init()`, `pthread_condattr_getpshared()`, `pthread_create()`, `pthread_mutex_init()`, `pthread_cond_init()`
**pthread_create()**

Create a thread

### Synopsis:

```c
#include <pthread.h>

int pthread_create( pthread_t* thread,
                    const pthread_attr_t* attr,
                    void* (*start_routine)(void* ),
                    void* arg );
```

### Arguments:

- **thread**
  - NULL, or a pointer to a `pthread_t` object where the function can store the thread ID of the new thread.

- **attr**
  - A pointer to a `pthread_attr_t` structure that specifies the attributes of the new thread. Instead of manipulating the members of this structure directly, use `pthread_attr_init()` and the `pthread_attr_set_*` functions. For the exceptions, see “QNX extensions,” below.
  
  If `attr` is NULL, the default attributes are used (see `pthread_attr_init()`).

  If you modify the attributes in `attr` after creating the thread, the thread’s attributes aren’t affected.

- **start_routine**
  - The routine where the thread begins, with `arg` as its only argument. If `start_routine()` returns, there’s an implicit call to `pthread_exit()`, using the return value of `start_routine()` as the exit status.

  The thread in which `main()` was invoked behaves differently. When it returns from `main()`, there’s an implicit call to `exit()`, using the return value of `main()` as the exit status.

- **arg**
  - The argument to pass to `start_routine`. 
Library:

```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_create()` function creates a new thread, with the attributes specified in the thread attribute object `attr`. The created thread inherits the signal mask of the parent thread, and its set of pending signals is empty.

- You must call `pthread_join()` or `pthread_detach()` for threads created with a `detachstate` attribute of `PTHREAD_CREATE_JOINABLE` (the default) before all of the resources associated with the thread can be released at thread termination.

- If you set the `stacksize` member of `attr`, the thread’s actual stack size is rounded up to a multiple of the system page size (which you can get by using the `_SC_PAGESIZE` constant in a call to `sysconf()`) if the system allocates the stack (the `stackaddr` member of `attr` is set to NULL). If the stack was previously allocated by the application, its size isn’t changed.

QNX extensions

If you adhere to the POSIX standard, there are some thread attributes that you can’t specify before creating the thread:

- You can’t disable cancellation for a thread.
- You can’t set the thread’s cancellation type.
- You can’t specify what happens when a signal is delivered to the thread.

There are no `pthread_attr_set_*` functions for these attributes.
As an QNX extension, you can OR the following bits into the __flags member of the pthread_attr_t structure before calling pthread_create():

PTHREAD_CANCEL_ENABLE
   Cancellation requests may be acted on according to the cancellation type (the default).

PTHREAD_CANCEL_DISABLE
   Cancellation requests are held pending.

PTHREAD_CANCELASYCHRONOUS
   If cancellation is enabled, new or pending cancellation requests may be acted on immediately.

PTHREAD_CANCEL_DEFERRED
   If cancellation is enabled, cancellation requests are held pending until a cancellation point (the default).

PTHREAD_MULTISIG_ALLOW
   Terminate all the threads in the process (the POSIX default).

PTHREAD_MULTISIG_DISALLOW
   Terminate only the thread that received the signal.

After creating the thread, you can change the cancellation properties by calling pthread_setcancelstate() and pthread_setcanceltype().

Returns:

EAGAIN Insufficient system resources to create thread.
EFAULT An error occurred trying to access the buffers or the start_routine provided.
EINVAL Invalid thread attribute object attr.
EOK Success.
Examples:

Create a thread in a detached state:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* function( void* arg )
{
    printf( "This is thread %d\n", pthread_self() );
    return( 0 );
}

int main( void )
{
    pthread_attr_t attr;

    pthread_attr_init( &attr );
    pthread_attr_setdetachstate(
        &attr, PTHREAD_CREATE_DETACHED );
    pthread_create( NULL, &attr, &function, NULL );

    /* Allow threads to run for 60 seconds. */
    sleep( 60 );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

pthread_attr_init(), pthread_exit(), pthread_setcancelstate(),
pthread_setcanceltype(), sysconf(), ThreadCreate()
**pthread_detach()**

Detach a thread from a process

**Synopsis:**

```c
#include <pthread.h>

int pthread_detach( pthread_t thread );
```

**Arguments:**

*thread*  
The ID of the thread that you want to detach, which you can get when you call `pthread_create()` or `pthread_self()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_detach()` function detaches the thread with the given ID from its process. When a detached thread terminates, all system resources allocated to that thread are immediately reclaimed.

**Returns:**

- **EOK**  
  Success.
- **EINVAL**  
  The thread *thread* is already detached.
- **ESRCH**  
  The thread *thread* doesn’t exist.

**Classification:**

POSIX 1003.1 THR

**Safety**

Cancellation point  
No
continued…
pthread_detach()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_join(), ThreadDetach().
**Synopsis:**

```c
#include <pthread.h>

int pthread_equal( pthread_t t1,
                   pthread_t t2 );
```

**Arguments:**

- `t1, t2` The thread IDs that you want to compare. You can get the IDs when you call `pthread_create()` or `pthread_self()`.

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_equal()` function compares the thread IDs of `t1` and `t2`. It doesn’t check to see if they’re valid thread IDs.

**Returns:**

- A nonzero value
  - The `t1` and `t2` thread IDs are equal.
- `0` The thread IDs aren’t equal.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

*continued…*
pthread_equal()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_create(), pthread_self()
**pthread_exit()**

Terminate a thread

**Synopsis:**
```c
#include <pthread.h>

void pthread_exit( void* value_ptr );
```

**Arguments:**
- `value_ptr` A pointer to a value that you want to be made available to any thread joining the thread that you’re terminating.

**Library:**
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_exit()` function terminates the calling thread. If the thread is joinable, the value `value_ptr` is made available to any thread joining the terminating thread (only one thread can get the return status). If the thread is detached, all system resources allocated to the thread are immediately reclaimed.

Before the thread is terminated, any cancellation cleanup handlers that have been pushed are popped and executed, and any thread-specific-data destructor functions are executed. Thread termination doesn’t implicitly release any process resources such as mutexes or file descriptors, or perform any process-cleanup actions such as calling `atexit()` handlers.

An implicit call to `pthread_exit()` is made when a thread other than the thread in which `main()` was first invoked returns from the start routine that was used to create it. The return value of the start routine is used as the thread’s exit status.
Don’t call `pthread_exit()` from cancellation-cleanup handlers or thread-specific-data destructor functions.

For the last process thread, `pthread_exit()` behaves as if you called `exit(0)`.

**Classification:**

POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atexit()`, `exit()`, `pthread_create()`, `pthread_cleanup_push()`, `pthread_cleanup_pop()`, `pthread_join()`, `ThreadDestroy()`.
Synopsis:

```c
#include <pthread.h>

int pthread_getconcurrency( void );
```

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

QNX doesn’t support the multiplexing of user threads on top of several kernel scheduled entities. As such, the `pthread_setconcurrency()` and `pthread_getconcurrency()` functions are provided for source code compatibility but they have no effect when called. To maintain the function semantics, the `new_level` parameter is saved when `pthread_setconcurrency()` is called so that a subsequent call to `pthread_getconcurrency()` returns the same value.

Returns:

The concurrency level set by a previous call to `pthread_setconcurrency()`, or 0 if there was no previous call.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

`pthread_setconcurrency()`
Thread-centric clocks are available for both real-time and general-purpose systems. They are provided in the form of clock IDs, which can be used to synchronize and coordinate activities across multiple threads or processes. 

The `pthread_getcpuclockid()` function is used to return the clock ID of the CPU-time clock from a specified thread.

### Synopsis:

```c
#include <sys/types.h>
#include <time.h>
#include <pthread.h>

extern int pthread_getcpuclockid(
    pthread_t id,
    clockid_t* clock_id);
```

### Arguments:

- **thread**
  The ID of the thread that you want to get the clock ID for, which you can get when you call `pthread_create()` or `pthread_self()`.

- **clock_id**
  A pointer to a `clockid_t` object where the function can store the clock ID.

### Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `pthread_getcpuclockid()` function returns the clock ID of the CPU-time clock of the thread specified by `id`, if the thread specified by `id` exists.

### Returns:

- **0**
  Success.

- **-1**
  An error occurred (`errno` is set).
pthread_getcpuclockid() © 2007, QNX Software Systems GmbH & Co. KG.

Errors:

ESRCH The value specified by id doesn’t refer to an existing thread.

Classification:

POSIX 1003.1 THR TCT

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

clock_getcpuclockid(), clock_getres(), clock_gettime(), ClockId(),
clock_settime(), pthread_getcpuclockid(), timer_create()
**pthread_getname_np()**

**Retrieve the name of a thread**

**Synopsis:**

```c
#include <pthread.h>

int pthread_getname_np(pthread_t tid, 
                       char* buffer, 
                       int buffersize);
```

**Arguments:**

- `tid` The ID of the thread whose name you want to get.
- `buffer` A storage place for the name.
- `buffersize` The size of the buffer, in bytes.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_getname_np()` function retrieves the name of the specified thread as a NULL-terminated string up to a maximum size (as determined by the `buffersize`). If the thread doesn’t have a name, the function stores a NULL-terminated empty string in the buffer. The “np” in the function’s name stands for “non-POSIX.”

This function was added in the QNX Neutrino Core OS 6.3.2.
**pthread_getname_np()** © 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

- **EOK**  Success.
- **E2BIG**  The name is larger than the accepted size.
- **EINVAL**  The name buffer length is invalid or smaller than the new name length.
- **EPERM**  The process doesn’t have superuser capabilities.

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*pthread_setname_np()*
Synopsis:

```c
#include <pthread.h>

int pthread_getschedparam(
   const pthread_t thread,
   int *policy,
   struct sched_param *param);
```

Arguments:

- `thread` The ID of the thread that you want to get the scheduling parameters for. You can get a thread ID by calling `pthread_create()` or `pthread_self()`.
- `policy` A pointer to a location where the function can store the scheduling policy; one of SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, or SCHED_OTHER.
- `param` A pointer to a `sched_param` structure where the function can store the scheduling parameters.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_getschedparam()` function gets the scheduling policy and associated scheduling parameters of thread `thread` and places them in `policy` and `param`.

Returns:

- `EOK` Success.
- `EFAULT` A fault occurred trying to access the buffers provided.
- `ESRCH` Invalid thread ID `thread`.
**Classification:**

POSIX 1003.1 THR TPS

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_create()`, `pthread_setschedparam()`, `sched_param`
*pthread_getspecific()* 
Get a thread-specific data value

Synopsis:

```c
#include <pthread.h>

void* pthread_getspecific( pthread_key_t key );
```

Arguments:

*key*  
The key associated with the data that you want to get. See *
 pthread_key_create()*.

Library:

*libc*  
Use the *-l c* option to *qcc* to link against this library. This library is usually included automatically.

Description:

The *pthread_getspecific()* function returns the thread-specific data value currently bound to the thread-specific-data key *key* in the calling thread, or NULL if no value is bound or the key doesn’t exist. You can call this function from a thread-specific-data destructor function.

You must call this function with a key that you got from *
 pthread_key_create()*.

You can’t use a key after destroying it with *
 pthread_key_delete()*.

Returns:

The data value, or NULL.

Examples:

See *pthread_key_create()*.
**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_key_create()`, `pthread_key_delete()`, `pthread_setspecific()`. 
Synopsis:

#include <pthread.h>

int pthread_join( pthread_t thread,
    void** value_ptr );

Arguments:

  thread       The target thread whose termination you’re waiting for.
  value_ptr    NULL, or a pointer to a location where the function can store the value passed to pthread_exit() by the target thread.

Library:

  libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The pthread_join() function blocks the calling thread until the target thread thread terminates, unless thread has already terminated. If value_ptr is non-NULL and pthread_join() returns successfully, then the value passed to pthread_exit() by the target thread is placed in value_ptr. If the target thread has been canceled then value_ptr is set to PTHREAD_CANCELED.

The non-POSIX pthread_timedjoin() function is similar to pthread_join(), except that an error is returned if the join doesn’t occur before a given time.

The target thread must be joinable. Multiple pthread_join(), pthread_timedjoin(), ThreadJoin(), and ThreadJoin_r() calls on the same target thread aren’t allowed. When pthread_join() returns successfully, the target thread has been terminated.
**pthread_join()**

Returns:

- **EOK** Success.
- **EBUSY** The thread `thread` is already being joined.
- **EDEADLK** The thread `thread` is the calling thread.
- **EFAULT** A fault occurred trying to access the buffers provided.
- **EINVAL** The thread `thread` isn’t joinable.
- **ESRCH** The thread `thread` doesn’t exist.

Classification:

- POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `pthread_create()`, `pthread_detach()`, `pthread_exit()`, `pthread_timedjoin()`, `ThreadJoin()`, `ThreadJoin_r()`
**Synopsis:**

```c
#include <pthread.h>

int pthread_key_create( pthread_key_t * key,
                       void (*destructor)( void * ) );
```

**Arguments:**

- **key**: A pointer to a `pthread_key_t` object where the function can store the new key.
- **destructor**: NULL, or a function to be called when you destroy the key.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_key_create()` function creates a thread-specific data key that's available to all threads in the process and binds an optional destructor function `destructor` to the key. If the function completes successfully the new key is returned in `key`.

Although the same key may be used by different threads, the values bound to the key using `pthread_setspecific()` are maintained on a per-thread basis and persist only for the lifetime of the thread.

When you create a key, the value NULL is bound with the key in all active threads. When you create a thread, the value NULL is bound to all defined keys in the new thread.

You can optionally associate a destructor function with each key value. At thread exit, if the key has a non-NULL destructor pointer, and the thread has a non-NULL value bound to the key, the destructor
function is called with the bound value as its only argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all destructor functions have been called for a thread, there are still non-NULL bound values, the destructor function is called repeatedly a maximum of PTHREAD_DESTRUCTOR_ITERATIONS times for each non-NULL bound value.

Returns:

- **EOK** Success.
- **EAGAIN** Insufficient system resources to create key or PTHREAD_KEYS_MAX has been exceeded.
- **ENOMEM** Insufficient memory to create key.

Examples:

This example shows how you can use thread-specific data to provide per-thread data in a thread-safe function:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

pthread_key_t buffer_key;

void buffer_key_destruct( void *value )
{
    free( value );
    pthread_setspecific( buffer_key, NULL );
}

char *lookup( void )
{
    char *string;

    string = (char *)pthread_getspecific( buffer_key );
    if( string == NULL ) {
        string = (char *)malloc( 32 );
        sprintf( string, "This is thread %d\n", pthread_self() );
        pthread_setspecific( buffer_key, (void *)string );
```
```c
void *function( void *arg )
{
    while( 1 )
    {
        puts( lookup() );
    }
    return( 0 );
}

int main( void )
{
    pthread_key_create( &buffer_key, 
                        &buffer_key_destruct );
    pthread_create( NULL, NULL, &function, NULL );

    /* Let the threads run for 60 seconds. */
    sleep( 60 );
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1 THR

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
Caveats:

The `pthread_key_create()` function is part of the POSIX 1003.1-1996 draft standard; its specific behavior may vary from system to system.

Before each destructor is called, the thread’s value for the corresponding key is set to NULL. Calling:

```c
pthread_setspecific(key, NULL);
```

in a key destructor isn’t required; this lets you use the same destructor for several keys.

See also:

`pthread_getspecific()`, `pthread_setspecific()`, `pthread_key_delete()`
**pthread_key_delete()**

Delete a thread-specific data key

**Synopsis:**

```c
#include <pthread.h>

int pthread_key_delete( pthread_key_t key );
```

**Arguments:**

- `key`  
  The key, which you created by calling `pthread_key_create()`, that you want to delete.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_key_delete()` function deletes the thread-specific data key `key` that you previously created with `pthread_key_create()`. The destructor function bound to `key` isn’t called by this function, and won’t be called at thread termination. You can call this function from a thread specific data destructor function.

If you need to release any data bound to the key in any threads, do so before deleting the key.

**Returns:**

- `EOK` Success.
- `EINVAL` Invalid thread-specific data key `key`.

**Classification:**

POSIX 1003.1 THR
**pthread_key_delete()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_key_create()`
**pthread_kill()**

Send a signal to a thread

Synopsis:

```c
#include <signal.h>

int pthread_kill( pthread_t thread, int sig );
```

Arguments:

- `thread` The ID of the thread that you want to send the signal to, which you can get when you call `pthread_create()` or `pthread_self()`.
- `sig` The signal that you want to send, or 0 if you just want to check for errors.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_kill()` function sends the signal `sig` to the thread `thread`. The target thread and calling thread must be in the same process. If `sig` is zero, error checking is performed but no signal is sent.

Returns:

- EOK Success.
- EAGAIN Insufficient system resources are available to deliver the signal.
- ESRCH Invalid thread ID `thread`.
- EINVAL Invalid signal number `sig`.
Classification:

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

kill(), ThreadDestroy()
**pthread_mutex_destroy()**

*Destroy a mutex*

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutex_destroy( pthread_mutex_t* mutex );
```

**Arguments:**

- `mutex` A pointer to the `pthread_mutex_t` object that you want to destroy.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_destroy()` function destroys the unlocked mutex `mutex`.

You can only destroy a locked mutex provided you’re the owner of that mutex.

Once you’ve destroyed a mutex, don’t reuse it without reinitializing it by calling `pthread_mutex_init()`.

**Returns:**

- **EOK** Success.
- **EBUSY** The `mutex` is locked by another thread.
- **EINVAL** Invalid mutex `mutex`.
Classification:

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_init(), SyncDestroy(), SyncMutexRevive()
**pthread_mutex_getprioceiling()**  
Get a mutex's priority ceiling

```c
#include <pthread.h>

int pthread_mutex_getprioceiling(
    const pthread_mutex_t* mutex,
    int* prioceiling);
```

**Arguments:**

- `mutex` A pointer to the `pthread_mutex_t` object that you want to priority ceiling for.
- `prioceiling` A pointer to a location where the function can store the priority ceiling.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_getprioceiling()` function returns the priority ceiling of `mutex` in `prioceiling`.

**Returns:**

- **EOK** Success.
- **EINVAL** The mutex specified by `mutex` doesn’t currently exist.
- **EPERM** The calling thread doesn’t have permission to get the priority ceiling.

**Classification:**

POSIX 1003.1 THR TPP
**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_mutex_getprioceiling()`
**pthread_mutex_init()**

Initialize mutex

**Synopsis:**

```c
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(
    pthread_mutex_t* mutex,
    const pthread_mutexattr_t* attr);
```

**Arguments:**

- `mutex` A pointer to the `pthread_mutex_t` object that you want to initialize.
- `attr` NULL, or a pointer to a `pthread_mutexattr_t` object that specifies the attributes that you want to use for the mutex. For more information, see `pthread_mutexattr_init()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_init()` function initializes the given mutex object, using the attributes specified by the mutex attributes object `attr`. If `attr` is NULL then the mutex is initialized with the default attributes (see `pthread_mutexattr_init()`). After initialization, the mutex is in an unlocked state.

You can initialize a statically allocated mutex with the default attributes by assigning to it the macro `PTHREAD_MUTEX_INITIALIZER` or `PTHREAD_RWLOCK_INITIALIZER` (for recursive mutexes).
**pthread_mutex_init()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

- **EOK** Success.
- **EAGAIN** All kernel synchronization objects are in use.
- **EBUSY** The given mutex was previously initialized and hasn’t been destroyed.
- **EFAULT** A fault occurred when the kernel tried to access `mutex` or `attr`.
- **EINVAL** The value specified by `attr` is invalid.

**Classification:**

- POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_mutexattr_init()`, `pthread_mutex_destroy()`, `SyncTypeCreate()`
**Synopsis:**

```c
#include <pthread.h>

int pthread_mutex_lock( pthread_mutex_t* mutex );
```

**Arguments:**

- `mutex` A pointer to the `pthread_mutex_t` object that you want to lock.

**Library:**

```c
#include <pthread.h>
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_lock()` function locks the mutex object referenced by `mutex`. If the mutex is already locked, then the calling thread blocks until it has acquired the mutex. When the function returns, the mutex object is locked and owned by the calling thread.

By default, if a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the higher-priority blocked thread waiting for the mutex. The owner returns to its real priority when it unlocks the mutex. For more information, see “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

If the mutex allows recursive behavior, a call to `pthread_mutex_lock()` while you own the mutex succeeds. You can allow recursive behavior by:

- statically initializing the mutex to `PTHREAD_RMUTEX_INITIALIZER`

Or:
• using `pthread_mutexattr_setrecursive()` to set the attribute to `PTHREAD_RECURSIVE_ALLOW` before calling `pthread_mutex_init()`.

If the mutex is recursive, you must call `pthread_mutex_unlock()` for each corresponding call to lock the mutex. The default POSIX behavior doesn’t allow recursive mutexes, and returns EDEADLK.

If a signal is delivered to a thread that’s waiting for a mutex, the thread resumes waiting for the mutex on returning from the signal handler.

Returns:

- **EOK** Success.
- **EAGAIN** Insufficient system resources available to lock the mutex.
- **EDEADLK** The calling thread already owns `mutex`, and the mutex doesn’t allow recursive behavior.
- **EFAULT** A fault occurred when the kernel tried to access the buffers you provided.
- **EINVAL** Invalid mutex `mutex`.
- **ETIMEDOUT** A kernel timeout unblocked the call.

Examples:

This example shows how you can use a mutex to synchronize access to a shared variable. In this example, `function1()` and `function2()` both attempt to access and modify the global variable `count`. Either thread could be interrupted between modifying `count` and assigning its value to the local `tmp` variable. Locking `mutex` prevents this from happening; if one thread has `mutex` locked, the other thread waits until it’s unlocked, before continuing.

```c
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```
int count = 0;

void* function1( void* arg )
{
    int tmp = 0;

    while( 1 )
    {
        pthread_mutex_lock( &mutex);
        tmp = count++;
        pthread_mutex_unlock( &mutex);
        printf( "Count is %d\n", tmp);
        /* snooze for 1 second */
        sleep( 1 );
    }
    return 0;
}

void* function2( void* arg )
{
    int tmp = 0;

    while( 1 )
    {
        pthread_mutex_lock( &mutex);
        tmp = count--;
        pthread_mutex_unlock( &mutex);
        printf( "** Count is %d\n", tmp);
        /* snooze for 2 seconds */
        sleep( 2 );
    }
    return 0;
}

int main( void )
{
    pthread_create( NULL, NULL, &function1, NULL);
    pthread_create( NULL, NULL, &function2, NULL);
    /* Let the threads run for 60 seconds. */
    sleep( 60 );
    return 0;
}
**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_mutex_init()`, `pthread_mutexattr_setrecursive()`,
`pthread_mutex_timedlock()`, `pthread_mutex_trylock()`,
`pthread_mutex_unlock()`, `SyncMutexLock()

“Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide
**pthread_mutex_setprioceiling()**

Set a mutex's priority ceiling

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutex_setprioceiling(
    pthread_mutex_t* mutex,
    int prioceiling,
    int* old_ceiling);
```

**Arguments:**

- `mutex` A pointer to the `pthread_mutex_t` object that you want to priority ceiling for.
- `prioceiling` The new value for the priority ceiling.
- `old_ceiling` A pointer to a location where the function can store the old value.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_setprioceiling()` function locks `mutex` (or blocks until it can lock it), changes its priority ceiling to `prioceiling`, and releases it. When the change is successful, the previous priority ceiling is returned in `old_ceiling`.

**Returns:**

- `EOK` Success.
- `EINVAL` The mutex specified by `mutex` doesn’t currently exist, or the priority requested by `prioceiling` is out of range.
- `EPERM` The calling thread doesn’t have permission to set the priority ceiling.
**Classification:**

POSIX 1003.1 THR TPP

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_mutex_setprioceiling()`
**Synopsis:**

```c
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(
    pthread_mutex_t * mutex,
    const struct timespec * abs_timeout );
```

**Arguments:**

- **mutex**
  - The mutex that you want to lock.

- **abs_timeout**
  - A pointer to a `timespec` structure that specifies the maximum time to wait to lock the mutex, expressed as an absolute time.

**Library:**

- **libc**
  - Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_timedlock()` function is called to lock the mutex object referenced by `mutex`. If the mutex is already locked, the calling thread blocks until the mutex becomes available as in the `pthread_mutex_lock()` function. If the mutex can’t be locked without waiting for another thread to unlock the mutex, the wait is terminated when the specified timeout expires.

By default, if a thread with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the higher-priority blocked thread waiting for the mutex. The owner returns to its real priority when it unlocks the mutex. For more information, see “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.
The timeout expires when the absolute time specified by `abs_timeout` passes, as measured by the clock on which timeouts are based (i.e., when the value of that clock equals or exceeds `abs_timeout`), or if the absolute time specified by `abs_timeout` has already been passed at the time of the call.

The timeout is based on the CLOCK_REALTIME clock. The `timespec` datatype is defined in the `<time.h>` header.

If the mutex can be locked immediately, the validity of the `abs_timeout` parameter isn’t checked, and the function won’t fail with a timeout.

As a consequence of the priority inheritance rules (for mutexes initialized with the PTHREAD_PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex is adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

Returns:

Zero on success, or an error number to indicate the error.

Errors:

- **EAGAIN**: The mutex couldn’t be acquired because the maximum number of recursive locks for the mutex has been exceeded.
- **EDEADLK**: The current thread already owns the mutex.
- **EINVAL**: The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread’s priority is higher than the mutex’ current priority ceiling; the process or thread would have blocked, and the `abs_timeout` parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million; or the value specified by `mutex` doesn’t refer to an initialized mutex object.
ETIMEDOUT The mutex couldn’t be locked before the specified timeout expired.

Classification:
POSIX 1003.1 THR TMO

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, `timespec`

“Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide
pthread_mutex_trylock() © 2007, QNX Software Systems GmbH & Co. KG.

Attempt to lock a mutex

Synopsis:

```c
#include <pthread.h>

int pthread_mutex_trylock( pthread_mutex_t* mutex );
```

Arguments:

- `mutex`: A pointer to the `pthread_mutex_t` object that you want to try to lock.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutex_trylock()` function attempts to lock the mutex `mutex`, but doesn’t block the calling thread if the mutex is already locked.

Returns:

- `EOK`: Success.
- `EAGAIN`: Insufficient resources available to lock the mutex.
- `EBUSY`: The `mutex` was already locked.
- `EINVAL`: Invalid mutex `mutex`.

Classification:

- POSIX 1003.1 THR
pthread_mutex_trylock()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_lock(), pthread_mutex_timedlock(),
pthread_mutex_unlock()

“Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the System Architecture guide
**pthread_mutex_unlock()** © 2007, QNX Software Systems GmbH & Co. KG.

**Unlock mutex**

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutex_unlock( pthread_mutex_t* mutex );
```

**Arguments:**

- `mutex`: A pointer to the `pthread_mutex_t` object that you want to unlock.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutex_unlock()` function unlocks the mutex `mutex`. The mutex should be owned by the calling thread. If there are threads blocked on the mutex then the highest priority waiting thread is unblocked and becomes the next owner of the mutex.

If the thread’s priority was raised when a higher-priority thread attempted to lock the mutex, the calling thread returns to its real priority when it unlocks the mutex. For more information, see “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

If `mutex` has been locked more than once, it must be unlocked the same number of times before the next thread is given ownership of the mutex. This only works for recursive mutexes.

**Returns:**

- `EOK`: Success.
- `EINVAL`: Invalid mutex `mutex`.
- `EPERM`: Current thread doesn’t own `mutex`. 
Classification:

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`pthread_mutex_lock()`, `pthread_mutex_timedlock()`,
`pthread_mutex_trylock()`, `SyncMutexUnlock()`

“Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the System Architecture guide
**pthread_mutexattr_destroy()**  © 2007, QNX Software Systems GmbH & Co.

Destroy mutex attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutexattr_destroy(
    pthread_mutexattr_t* attr);
```

**Arguments:**

- `attr` A pointer to the `pthread_mutexattr_t` object that you want to destroy.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutexattr_destroy()` function destroys the mutex attribute object `attr`.

Once you’ve destroyed a mutex attribute object, don’t reuse it without reinitializing it by calling `pthread_mutexattr_init()`.

**Returns:**

- `EOK` Success.
- `EINVAL` Invalid mutex attribute object `attr`.

**Classification:**

- POSIX 1003.1 THR
###(pthread_mutexattr_destroy())

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

### See also:

`pthread_mutexattr_init(), pthread_mutex_init()`
Get the priority ceiling of a mutex attribute object

Synopsis:

```c
#include <pthread.h>

int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t* attr,
    int* prioceiling);
```

Arguments:

- `attr`: A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.
- `prioceiling`: A pointer to a location where the function can store the priority ceiling.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getprioceiling()` function sets `prioceiling` to the current mutex attribute `attr`'s scheduling priority ceiling. The mutex attribute object `attr` must have been previously created with `pthread_mutexattr_init()`.

Returns:

- `EOK`: Success.
- `EINVAL`: Invalid value specified by `attr` or `prioceiling`.
- `EPERM`: The caller doesn’t have the privilege to perform the operation.
**Classification:**

POSIX 1003.1 THR TPP

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

pthread_cond_init(), pthread_create(), pthread_mutex_init(),
pthread_mutexattr_getprotocol(), pthread_mutexattr_getrecursive(),
pthread_mutexattr_setprioceiling(),
pthread_mutexattr_setrecursive()
Get a mutex’s scheduling protocol

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutexattr_getprotocol(
    pthread_mutexattr * attr,
    int * protocol);
```

**Arguments:**

- `attr` A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.
- `protocol` A pointer to a location where the function can store the scheduling protocol.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutexattr_getprotocol()` function sets `protocol` to the current mutex attribute `attr`’s scheduling protocol. The structure pointed to by `attr` must have been previously created with `pthread_mutexattr_init()`.

The `protocol` attribute defines the protocol for using mutexes:

Currently, `protocol` may be set to:

**PTHREAD_PRIO_INHERIT**

When a thread is blocking higher-priority threads by locking one or more mutexes with this attribute, the thread’s priority is raised to that of the highest priority thread waiting on the `PTHREAD_PRIO_INHERIT` mutex.
PTHREAD_PRIO_PROTECT

The thread executes at the highest priority or priority ceilings of all the mutexes owned by the thread and initialized with PTHREAD_PRIO_PROTECT, whether other threads are blocked or not.

A thread holding a PTHREAD_PRIO_INHERIT mutex won’t be moved to the tail of the scheduling queue if its original priority is changed (by a call to `pthread_schedsetparam()`, for example). This remains true if the thread unlocks the PTHREAD_PRIO_INHERIT mutex.

The POSIX protocol of PTHREAD_PRIONONE isn’t currently supported.

Returns:

- EOK  Success.
- EINVAL  Invalid mutex attribute `attr`.

Classification:

POSIX 1003.1 THR TPP | TPI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`pthread_mutexattr_setprotocol()`, `pthread_mutexattr_setrecursive()`

“Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide
**pthread_mutexattr_getpshared()**

Get the process-shared attribute from a mutex attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutexattr_getpshared(
    const pthread_mutexattr_t* attr,
    int* pshared);
```

**Arguments:**

- `attr` A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.
- `pshared` A pointer to a location where the function can store the process-shared attribute.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutexattr_getpshared()` function gets the process-shared attribute from the mutex attribute object `attr` and stores it in the memory pointed to by `pshared`.

If the process-shared attribute is set to `PTHREAD_PROCESS_SHARED`, any thread that has access to the memory where the condition variable is allocated can operate on it, even if the condition variable is allocated in memory that’s shared by multiple processes.

If the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`, the condition variable can only be accessed by threads created within the same process as the thread that initialized the condition variable; if threads from other processes try to access the `PTHREAD_PROCESS_PRIVATE` condition variable, the behavior is undefined. The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`. 
pthread_mutexattr_getpshared() © 2007, QNX Software Systems GmbH & Co. KG.

Returns:

EOK    Success.
EINVAL The mutex attribute object specified by attr is invalid.

Classification:

POSIX 1003.1 THR TSH

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_init(), pthread_create(), pthread_mutex_init(),
pthread_mutexattr_setpshared(), pthread_mutexattr_setrecursive()
pthread_mutexattr_getrecursive()
Get the recursive attribute from a mutex attribute object

Synopsis:

```c
#include <pthread.h>

int pthread_mutexattr_getrecursive(
    const pthread_mutexattr_t* attr,
    int* recursive);
```

Arguments:

- `attr` A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.
- `recursive` A pointer to a location where the function can store the recursive attribute.

Library:

`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_getrecursive()` function gets the recursive attribute from the mutex attribute object `attr` and stores it in `recursive`.

If the recursive attribute is set to `PTHREAD_RECURSIVE_ENABLE`, a thread that has already locked the mutex can lock it again without blocking. If the recursive attribute is set to `PTHREAD_RECURSIVE_DISABLE`, any thread that tries to lock the mutex will block, if that mutex is already locked.

The default value of the recursive attribute is `PTHREAD_RECURSIVE_DISABLE`.

Returns:

- `EOK` Success.
- `EINVAL` Invalid mutex attribute object `attr`.
**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_mutexattr_init()`, `pthread_mutexattr_setrecursive()`
Synopsis:

```c
#include <pthread.h>

int pthread_mutexattr_gettype(
    const pthread_mutexattr_t * attr,
    int * type);
```

Arguments:

- `attr` A pointer to the `pthread_mutexattr_t` object that you want to get the attribute from.
- `type` A pointer to a location where the function can store the type.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_gettype()` function gets the mutex type attribute in the `type` parameter. Valid mutex types include:

- **PTHREAD_MUTEX_NORMAL**
  - No deadlock detection. A thread that attempts to relock this mutex without first unlocking it deadlocks. Attempts to unlock a mutex locked by a different thread or attempts to unlock an unlocked mutex result in undefined behavior.

- **PTHREAD_MUTEX_ERRORCHECK**
  - Provides error checking. A thread returns with an error when it attempts to:
    - Relock this mutex without first unlocking it.
    - Unlock a mutex that another thread has locked.
    - Unlock an unlocked mutex.
PTHREAD_MUTEX_RECURSIVE
A thread that attempts to relock this mutex without first unlocking it succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type PTHREAD_MUTEX_NORMAL can’t occur with this mutex type. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread that attempts to unlock a mutex that another thread has locked, or unlock an unlocked mutex, returns with an error.

PTHREAD_MUTEX_DEFAULT
The default value of the type attribute. Attempts to recursively lock a mutex of this type, or unlock a mutex of this type that isn’t locked by the calling thread, or unlock a mutex of this type that isn’t locked, results in undefined behavior.

Returns:
Zero, and the value of the type attribute of attr is stored in the object referenced by the type parameter; otherwise, an error.

Errors:
EINVVAL Invalid value specified by attr.

Classification:
POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

An application shouldn’t use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a `pthread_cond_wait()` or `pthread_cond_timedwait()` may not actually release the mutex (if it’s been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

See also:

`pthread_cond_timedwait()`, `pthread_cond_wait()`,
`pthread_mutexattr_settype()`
thread_mutexattr_init() © 2007, QNX Software Systems GmbH & Co. KG.

Initialize a mutex attribute object

Synopsis:

```
#include <pthread.h>

int pthread_mutexattr_init(
    const pthread_mutexattr_t* attr
);
```

Arguments:

*attr* A pointer to the `pthread_mutexattr_t` object that you want to initialize.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_init()` function initializes the attributes in the mutex attribute object `attr` to their default values. After initializing a mutex attribute object, you can use it to initialize one or more mutexes by calling `pthread_mutex_init()`.

The mutex attributes and their default values are:

* protocol
  PTHREAD_PRIO_INHERIT—when a thread is blocking higher-priority threads by locking one or more mutexes with this attribute, the thread’s priority is raised to that of the highest priority thread waiting on the PTHREAD_PRIO_INHERIT mutex.

* recursive
  PTHREAD_RECURSIVE_DISABLE—threads can’t recursively lock a mutex; any thread that tries to lock an already locked mutex becomes blocked.

After calling this function, you can use the `pthread_mutexattr_*` family of functions to make any changes to the attributes:
**pthread_mutexattr_init()**

<table>
<thead>
<tr>
<th>Get</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pthread_mutexattr_getprioceiling()</code></td>
<td><code>pthread_mutexattr_setprioceiling()</code></td>
</tr>
<tr>
<td><code>pthread_mutexattr_getprotocol()</code></td>
<td><code>pthread_mutexattr_setprotocol()</code></td>
</tr>
<tr>
<td><code>pthread_mutexattr_getpshared()</code></td>
<td><code>pthread_mutexattr_setpshared()</code></td>
</tr>
<tr>
<td><code>pthread_mutexattr_getrecursive()</code></td>
<td><code>pthread_mutexattr_setrecursive()</code></td>
</tr>
<tr>
<td><code>pthread_mutexattr_gettype()</code></td>
<td><code>pthread_mutexattr_settype()</code></td>
</tr>
</tbody>
</table>

**Returns:**

- **EOK** Success.
- **ENOMEM** Insufficient memory to initialize mutex attribute object `attr`.

**Classification:**

- POSIX 1003.1 THR

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `pthread_mutex_init()`, `pthread_mutexattr_destroy()`,
- `pthread_mutexattr_getprioceiling()`,
- `pthread_mutexattr_getprotocol()`, `pthread_mutexattr_getpshared()`, `pthread_mutexattr_getrecursive()`, `pthread_mutexattr_gettype()`,
- `pthread_mutexattr_setprioceiling()`. 
`pthread_mutexattr_init()`

`pthread_mutexattr_setprotocol()`, `pthread_mutexattr_setpshared()`, `pthread_mutexattr_setrecursive()`, `pthread_mutexattr_settype()`

“Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide
**pthread_mutexattr_setprioceiling()**
Set the priority ceiling of a mutex attribute object

**Synopsis:**
```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(
    pthread_mutexattr_t* attr,
    int prioceiling);
```

**Arguments:**
- `attr` A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.
- `prioceiling` The new value for the priority ceiling.

**Library:**
`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `pthread_mutexattr_setprioceiling()` function sets the mutex attribute `attr`’s scheduling priority ceiling to `prioceiling`. Note that `attr` must have been previously created with `pthread_mutexattr_init()`.

**Returns:**
- `EOK` Success.
- `EINVAL` Invalid value specified by `attr` or `prioceiling`.
- `EPERM` The caller doesn’t have the privilege to perform the operation.

**Classification:**
POSIX 1003.1 THR TPP
pthread_mutexattr_setprioceiling() © 2007, QNX Software Systems GmbH & Co. KG.

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

 pthread_create(), pthread_mutex_init(), pthread_cond_init(),
 pthread_mutexattr_getprioceiling(),
 pthread_mutexattr_getprotocol(), pthread_mutexattr_getrecursive()
**pthread_mutexattr_setprotocol()**

Set a mutex's scheduling protocol

### Synopsis:

```c
#include <pthread.h>

int pthread_mutexattr_setprotocol(
    pthread_mutexattr * attr,
    int protocol);
```

### Arguments:

- **attr**
  A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

- **protocol**
  The new value of the scheduling protocol; one of:
  - `PTHREAD_PRIO_INHERIT` — when a thread is blocking higher-priority threads by locking one or more mutexes with this attribute, raise the thread’s priority to that of the highest priority thread waiting on the PTHREAD_PRIO_INHERIT mutex.
  - `PTHREAD_PRIO_PROTECT` — execute the thread at the highest priority or priority ceilings of all the mutexes owned by the thread and initialized with PTHREAD_PRIO_PROTECT, whether other threads are blocked or not.

---

The POSIX protocol of `PTHREAD_PRIO_NONE` isn’t currently supported.

### Library:

- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**Description:**

The `pthread_mutexattr_setprotocol()` function sets the mutex attribute `attr`’s scheduling protocol to `protocol`. The structure pointed to by `attr` must have been previously created with `pthread_mutexattr_init()`. The `protocol` attribute defines the protocol for using mutexes. A thread holding a PTHREAD_PRIO_INHERIT mutex won’t be moved to the tail of the scheduling queue if its original priority is changed (by a call to `pthread_s schedsetparam()`, for example). This remains true if the thread unlocks the PTHREAD_PRIO_INHERIT mutex.

**Returns:**

- **EOK** Success.
- **ENOTSUP** The `protocol` argument is an unsupported or an invalid value.

**Classification:**

<table>
<thead>
<tr>
<th>POSIX 1003.1 THR TPP</th>
<th>TPI</th>
</tr>
</thead>
</table>

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

- `pthread_mutexattr_getprotocol()`, `pthread_mutexattr_getrecursive()`
- “Mutexes: mutual exclusion locks” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide
#include <pthread.h>

int pthread_mutexattr_setpshared(
    pthread_mutexattr_t* attr,
    int pshared );

Arguments:

attr      A pointer to the pthread_mutexattr_t object that you want to set the attribute in.

pshared   The new value of the process-shared attribute; one of:

- PTHREAD_PROCESS_SHARED — any thread that has access to the memory where the mutex is allocated can operate on it, even if the mutex is allocated in memory that's shared by multiple processes.
- PTHREAD_PROCESS_PRIVATE — the mutex can be accessed only by threads created within the same process as the thread that initialized the mutex; if threads from other processes try to access the PTHREAD_PROCESS_PRIVATE mutex, the behavior is undefined.

The default value of the process-shared attribute is PTHREAD_PROCESS_PRIVATE.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The pthread_mutexattr_setpshared() function sets the process-shared attribute in a mutex attribute object, attr, to the value given by pshared.
Returns:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid mutex attribute object, <em>attr</em>.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The new value specified in <em>pshared</em> isn’t PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.</td>
</tr>
</tbody>
</table>

Classification:

POSIX 1003.1 THR TSH

<table>
<thead>
<tr>
<th>Safety Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

*pthread_cond_init(), pthread_create(),
 pthread_mutexattr_getpshared(), pthread_mutexattr_getrecursive(),
 pthread_mutex_init(), pthread_mutexattr_setrecursive()
**Synopsis:**

```c
#include <pthread.h>

int pthread_mutexattr_setrecursive(
    pthread_mutexattr_t* attr,
    int recursive);
```

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>attr</code></td>
<td>A pointer to the <code>pthread_mutexattr_t</code> object that you want to set the attribute in.</td>
</tr>
<tr>
<td><code>recursive</code></td>
<td>The new value for the recursive attribute; one of:</td>
</tr>
<tr>
<td></td>
<td>- <code>PTHREAD_RECURSIVE_ENABLE</code> — a thread that has already locked the mutex can lock it again without blocking.</td>
</tr>
<tr>
<td></td>
<td>- <code>PTHREAD_RECURSIVE_DISABLE</code> — any thread that tries to lock the mutex blocks, if that mutex is already locked.</td>
</tr>
</tbody>
</table>

The default value of the recursive attribute is `PTHREAD_RECURSIVE_DISABLE`.

**Library:**

```
libc
```

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_mutexattr_setrecursive()` function sets the recursive attribute in a mutex attribute object, `attr`. 
**pthread_mutexattr_setrecursive()** © 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

- **EOK** Success.
- **EINVAL** Invalid mutex attribute object, `attr`, or the value specified by `recursive` isn’t `PTHREAD_RECURSIVE_ENABLE` or `PTHREAD_RECURSIVE_DISABLE`.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_mutexattr_getrecursive()`, `pthread_mutexattr_init()`
pthread_mutexattr_settype()

Set a mutex type

Synopsis:

```c
#include <pthread.h>

int pthread_mutexattr_settype(
    pthread_mutexattr_t * attr,
    int type);
```

Arguments:

- **attr** A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.
- **type** The new type; one of:
  - PTHREAD_MUTEX_NORMAL — no deadlock detection. A thread that attempts to relock this mutex without first unlocking it deadlocks. Attempts to unlock a mutex locked by a different thread or attempts to unlock an unlocked mutex result in undefined behavior.
  - PTHREAD_MUTEX_ERRORCHECK — provides error checking. A thread returns with an error when it attempts to relock this mutex without first unlocking it, unlock a mutex that another thread has locked, or unlock an unlocked mutex.
  - PTHREAD_MUTEX_RECURSIVE — a thread that attempts to relock this mutex without first unlocking it succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type PTHREAD_MUTEX_NORMAL can’t occur with this mutex type. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread that attempts to unlock a mutex that another thread has locked, or unlock an unlocked mutex, returns with an error.
  - PTHREAD_MUTEX_DEFAULT — the default value of the `type` attribute. Attempts to recursively lock a mutex of this type, or unlock a mutex of this type that isn’t locked by
the calling thread, or unlock a mutex of this type that isn’t locked, results in undefined behavior.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_mutexattr_settype()` function sets the mutex type attribute in the `type` parameter.

Returns:

Zero for success, or an error number.

Errors:

```
EINVAL The value specified by `attr` or `type` is invalid.
```

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

An application shouldn’t use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a `pthread_cond_wait()` or `pthread_cond_timedwait()` may not
pthread_mutexattr_settype()

actually release the mutex (if it’s been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

See also:

pthread_cond_timedwait(), pthread_cond_wait(),
pthread_mutexattr_gettype()
**pthread_once()**

Dynamic package initialization

**Synopsis:**

```c
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once( pthread_once_t* once_control,
                  void (*init_routine)(void) );
```

**Arguments:**

- `once_control`: A pointer to a `pthread_once_t` object that the function uses to determine whether or not to run the initialization code.

You must set the `pthread_once_t` object to the macro `PTHREAD_ONCE_INIT` before using it for the first time.

- `init_routine`: The function that you want to call to do any required initialization.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_once()` function uses the once-control object `once_control` to determine whether the initialization routine `init_routine` should be called. The first call to `pthread_once()` by any thread in a process, with a given `once_control`, calls `init_routine` with no arguments. Subsequent calls of `pthread_once()` with the same `once_control` won’t call `init_routine`.
No thread will execute past this function until the `init_routine` returns.

**Returns:**

- **EOK**  
  Success.
- **EINVAL**  
  Uninitialized once-control object `once_control`.

**Examples:**

This example shows how you can use once-initialization to initialize a library; both `library_entry_point1()` and `library_entry_point2()` need to initialize the library, but that needs to happen only once:

```c
#include <stdio.h>
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

void library_init( void )
{
    /* initialize the library */
}

void library_entry_point1( void )
{
    pthread_once( &once_control, library_init );
    /* do stuff for library_entry_point1... */
}

void library_entry_point2( void )
{
    pthread_once( &once_control, library_init );
    /* do stuff for library_entry_point1... */
}
```

This initializes the library once; if multiple threads call `pthread_once()`, only one actually enters the `library_init()` function. The other threads block at the `pthread_once()` call until `library_init()` has returned. The `pthread_once()` function also ensures that
**pthread_once()**  © 2007, QNX Software Systems GmbH & Co. KG.

`library_init()` is only ever called once; subsequent calls to the library entry points skip the call to `library_init()`.

**Classification:**

POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
**pthread_rwlock_destroy()**

Destroy a read-write lock

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlock_destroy( pthread_rwlock_t* rwl );
```

**Arguments:**

- **rwl** A pointer to a `pthread_rwlock_t` object that you want to destroy.

**Library:**

`libc`

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_destroy()` function destroys the read-write lock referenced by `rwl`, and releases the system resources used by the lock. You can destroy the read-write lock if one of the following is true:

- no thread has a active shared or exclusive lock on `rwl`
- the calling thread has an active exclusive lock on `rwl`.

After successfully destroying a read-write lock, don’t use it again without reinitializing it by calling `pthread_rwlock_init()`.

**Returns:**

- **EOK** Success.
- **EBUSY** The read-write lock `rwl` is still in use. The calling thread doesn’t have an exclusive lock.
**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

pthread_rwlock_init(), pthread_rwlock_rdlock(),
pthread_rwlock_tryrdlock(), pthread_rwlock_trywrlock(),
pthread_rwlock_trywrlock(),
pthread_rwlock_unlock(), pthread_rwlock_wrlock()
**pthread_rwlock_init()**

Initialize a read-write lock

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlock_init(
    pthread_rwlock_t * rwl,
    const pthread_rwlockattr_t * attr );
```

**Arguments:**

- `rwl` A pointer to a `pthread_rwlock_t` object that you want to initialize.
- `attr` NULL, or a pointer to a `pthread_rwlockattr_t` object that specifies the attributes you want to use for the read-write lock; see `pthread_rwlockattr_init()`.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_init()` function initializes the read-write lock referenced by `rwl` with the attributes of `attr`. You must initialize read-write locks before using them. If `attr` is NULL, `rwl` is initialized with the default values for the attributes.

Following a successful call to `pthread_rwlock_init()`, the read-write lock is unlocked, and you can use it in subsequent calls to `pthread_rwlock_destroy()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, and `pthread_rwlock_wrlock()`. This lock remains usable until you destroy it by calling `pthread_rwlock_destroy()`.

If the read-write lock is statically allocated, you can initialize it with the default values by setting it to `PTHREAD_RWLOCK_INITIALIZER`. 
More than one thread may hold a shared lock at any time, but only one thread may hold an exclusive lock. This avoids reader and writer starvation during frequent contention by:

- favoring blocked readers over writers after a writer has just released an exclusive lock, and

- favoring writers over readers when there are no blocked readers.

Under heavy contention, the lock alternates between a single exclusive lock followed by a batch of shared locks.

**Returns:**

- **EOK** Success.
- **EAGAIN** Insufficient system resources to initialize the read-write lock.
- **EBUSY** The read-write lock `rwl` has been initialized or unsuccessfully destroyed.
- **EFAULT** A fault occurred when the kernel tried to access `rwl` or `attr`.
- **EINVAL** Invalid read-write lock attribute object `attr`.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
Caveats:

Beware of *priority inversion* when using read-write locks. A high-priority thread may be blocked waiting on a read-write lock locked by a low-priority thread.

The microkernel has no knowledge of read-write locks, and therefore can’t boost the low-priority thread to prevent the priority inversion.

See also:

`pthread_rwlockattr_init()`, `pthread_rwlock_destroy()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_wrlock()`, `pthread_rwlock_unlock()`
**pthread_rwlock_rdlock()** © 2007, QNX Software Systems GmbH & Co. KG.

Acquire a shared read lock on a read-write lock

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlock_rdlock( pthread_rwlock_t* rwl );
```

**Arguments:**

- `rwl` A pointer to a `pthread_rwlock_t` object that you want to lock for reading.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_rdlock()` function acquires a shared lock on the read-write lock referenced by `rwl`. If the read-write lock is already exclusively locked, the calling thread blocks until the exclusive lock is released.

If a signal is delivered to a thread waiting to lock a read-write lock, it will resume waiting for the lock after returning from the signal handler.

A thread may hold several read locks on the same read-write lock; it must call `pthread_rwlock_unlock()` multiple times to release its read lock.

**Returns:**

- **EOK** Success.
- **EAGAIN** On the first use of statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
**pthread_rwlock_rdlock()**

EDEADLK   The calling thread already has an exclusive lock for `rwl`.
EFAULT    A fault occurred when the kernel tried to access `rwl`.
EINVVAL   The read-write lock `rwl` is invalid.

**Classification:**

POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_rwlock_destroy()`, `pthread_rwlock_init()`, `pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_unlock()`, `pthread_rwlock_wrlock()`
Lock a read-write lock for reading

Synopsis:

```c
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(
    pthread_rwlock_t * rwlock,
    const struct timespec * abs_timeout);
```

Arguments:

- `rwlock` The read-write lock that you want to lock.
- `abs_timeout` A pointer to a `timespec` that specifies the maximum time to wait to acquire the lock, expressed as an absolute time.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlock_timedrdlock()` function applies a read lock to the read-write lock referenced by `rwlock` as in `pthread_rwlock_rdlock()`.

However, if the lock can’t be acquired without waiting for other threads to unlock it, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by `abs_timeout` passes, as measured by the clock on which timeouts are based (i.e. when the value of that clock equals or exceed `abs_timeout`), or if the absolute time specified by `abs_timeout` has already been passed at the time of the call.

The timeout is based on the `CLOCK_REALTIME` clock.

If the read-write lock can be locked immediately, the validity of the `abs_timeout` parameter isn’t checked, and the function won’t fail with a timeout.
If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to `pthread_rwlock_timedrdlock()`, upon return from the signal handler the thread resumes waiting for the lock as if it hadn’t been interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on `rwlock`. The results are undefined if this function is called with an uninitialized read-write lock.

**Returns:**

Zero if the lock for reading on the read-write lock object referenced by `rwlock` is acquired, or an error number to indicate the error.

**Errors:**

- **EAGAIN**  
  Couldn’t acquire read lock because the maximum number of read locks for lock would be exceeded.

- **EDEADLK**  
  The calling thread already holds a write lock on `rwlock`.

- **EINVAL**  
  The value specified by `rwlock` doesn’t refer to an initialized read-write lock object, or the `abs_timeout` nanosecond value is less than zero or greater than or equal to 1,000 million.

- **ETIMEDOUT**  
  The lock couldn’t be acquired before the specified timeout expired.

**Classification:**

POSIX 1003.1 THR TMO

**Safety**

- Cancellation point  Yes
- Interrupt handler  No

*continued…*
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_rwlock_destroy()`, `pthread_rwlock_init()`, `pthread_rwlock_timedwrlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_tryrdlock()`, `pthread_rwlock_unlock()`, `pthread_rwlock_wrlock()`, `timespec`
**Synopsis:**

```c
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(
    pthread_rwlock_t * rwlock,
    const struct timespec * abs_timeout);
```

**Arguments:**

- `rwlock` The read-write lock that you want to lock.
- `abs_timeout` A pointer to a `timespec` that specifies the maximum time to wait to acquire the lock, expressed as an absolute time.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_timedwrlock()` function applies a write lock to the read-write lock referenced by `rwlock` as in `pthread_rwlock_wrlock()`.

However, if the lock can’t be acquired without waiting for other threads to unlock the lock, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by `abs_timeout` passes, as measured by the clock on which timeouts are based (i.e. when the value of that clock equals or exceed `abs_timeout`), or if the absolute time specified by `abs_timeout` has already been passed at the time of the call.

The timeout is based on the `CLOCK_REALTIME` clock.

If the read-write lock can be locked immediately, the validity of the `abs_timeout` parameter isn’t checked, and the function won’t fail with a timeout.
If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to `pthread_rwlock_timedwrlock()`, upon return from the signal handler the thread resumes waiting for the lock as if it hadn’t been interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on `rwlock`. The results are undefined if this function is called with an uninitialized read-write lock.

Returns:

Zero if the lock for writing on the read-write lock object referenced by `rwlock` is acquired, or an error number to indicate the error.

Errors:

EAGAIN  
Could’t acquire read lock because the maximum number of read locks for lock would be exceeded.

EDEADLK  
The calling thread already holds the `rwlock`.

EINVAL  
The value specified by `rwlock` doesn’t refer to an initialized read-write lock object, or the `abs_timeout` nanosecond value is less than zero or greater than or equal to 1,000 million.

ETIMEDOUT  
The lock couldn’t be acquired before the specified timeout expired.

Classification:

POSIX 1003.1 THR TMO

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
See also:

pthread_rwlock_destroy(), pthread_rwlock_init(),
pthread_rwlock_timedrdlock(), pthread_rwlock_trywrlock(),
pthread_rwlock_tryrdlock(), pthread_rwlock_unlock(),
pthread_rwlock_wrlock(), timespec
**pthread_rwlock_tryrdlock()** © 2007, QNX Software Systems GmbH & Co. KG.

*Attempt to acquire a shared lock on a read-write lock*

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlock_tryrdlock(
    pthread_rwlock_t* rwl );
```

**Arguments:**

- `rwl` A pointer to a `pthread_rwlock_t` object that you want to lock for reading.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_tryrdlock()` function attempts to acquire a shared lock on the read-write lock referenced by `rwl`. If the read-write lock is already exclusively locked by any thread (including the calling thread), the function returns immediately instead of blocking until a read lock can be obtained.

**Returns:**

- **EOK** Success.
- **EAGAIN** On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
- **EBUSY** The read-write lock was already write locked.
- **EDEADLK** The calling thread already has an exclusive lock for `rwl`.
- **EFAULT** A fault occurred when the kernel tried to access `rwl`.
- **EINVAL** The read-write lock `rwl` is invalid.
**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_rwlock_destroy()`, `pthread_rwlock_init()`,
`pthread_rwlock_rdlock()`, `pthread_rwlock_trywrlock()`,
`pthread_rwlock_unlock()`, `pthread_rwlock_wrlock()`
**pthread_rwlock_trywrlock()** © 2007, QNX Software Systems GmbH & Co. KG.

Attempt to acquire an exclusive lock on a read-write lock

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlock_trywrlock(
    pthread_rwlock_t* rw);
```

**Arguments:**

- `rwl` A pointer to a `pthread_rwlock_t` object that you want to lock for writing.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_trywrlock()` function attempts to acquire an exclusive lock on the read-write lock referenced by `rwl`. If the read-write lock is already exclusively locked or shared locked, the function returns immediately instead of blocking until an exclusive lock can be obtained.

The function may need to block to determine the state of the read-write lock.

**Returns:**

- **EOK** Success.
- **EAGAIN** On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
- **EBUSY** The read-write lock was already write locked or read locked.
EDEADLK | The calling thread already has an exclusive lock for `rwl`
EFAULT | A fault occurred when the kernel tried to access `rwl`
EINVAL | The read-write lock `rwl` is invalid.

**Classification:**

POSIX 1003.1 THR

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_rwlock_destroy()`, `pthread_rwlock_init()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_tryrdlock()`, `pthread_rwlock_unlock()`, `pthread_rwlock_wrlock()`
**pthread_rwlock_unlock()**

**Unlock a read-write lock**

### Synopsis:

```c
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t* rwl);
```

### Arguments:

- `rwl`  
  A pointer to a `pthread_rwlock_t` object that you want to unlock.

### Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `pthread_rwlock_unlock()` function unlocks a read-write lock referenced by `rwl`. The read-write lock may become available for any threads that were blocked on the read-write lock, depending on whether the read-write lock had been locked in exclusive or shared mode.

The read-write lock should be owned by the calling thread. If the calling thread doesn’t hold the lock, no error status is returned, and the behavior of this read-write lock is now undefined.

### Returns:

- **EOK**  
  Success.

- **EAGAIN**  
  On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.

- **EFAULT**  
  A fault occurred when the kernel tried to access `rwl`. 

---

2320  
C Library — P to R  
September 10, 2007


**pthread_rwlock_unlock()**

- **EINVAL** The read-write lock `rwl` is invalid.
- **EPERM** No thread has a read or write lock on `rwl` or the calling thread doesn’t have a write lock on `rwl`.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

- `pthread_rwlock_destroy()`, `pthread_rwlock_init()`,
- `pthread_rwlock_rdlock()`, `pthread_rwlock_tryrdlock()`,
- `pthread_rwlock_trywrlock()`, `pthread_rwlock_wrlock()`
**pthread_rwlock_wrlock()** © 2007, QNX Software Systems GmbH & Co. KG.

Acquire an exclusive write lock on a read-write lock

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlock_wrlock(
    pthread_rwlock_t* rwl
);
```

**Arguments:**

- **rwl** A pointer to a `pthread_rwlock_t` object that you want to lock for writing.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlock_wrlock()` function acquires an exclusive lock on the read-write lock referenced by `rwl`. If the read-write lock is already shared-locked by any thread (including the calling thread) or exclusively-locked by any thread (other than the calling thread), the calling thread blocks until all shared locks and exclusive locks are released.

If a signal is delivered to a thread waiting to lock a read-write lock, it resumes waiting for the lock after returning from the signal handler.

**Returns:**

- **EOK** Success.
- **EAGAIN** On the first use of a statically initialized read-write lock, insufficient system resources existed to initialize the read-write lock.
- **EDEADLK** The calling thread already has an exclusive lock for `rwl`. 
pthread_rwlock_wrlock()

EFAULT    A fault occurred when the kernel tried to access rwl.
EINVAL    The read-write lock rwl is invalid.

Classification:
POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

pthread_rwlock_destroy(), pthread_rwlock_init(),
pthread_rwlock_rdlock(), pthread_rwlock_tryrdlock(),
pthread_rwlock_trywrlock(), pthread_rwlock_unlock()
Destroy a read-write lock attribute object

**Synopsis:**

```
#include <pthread.h>

int pthread_rwlockattr_destroy(
    pthread_rwlockattr_t* attr
);
```

**Arguments:**

- `attr` A pointer to the `pthread_rwlockattr_t` object that you want to destroy.

**Library:**

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlockattr_destroy()` function destroys a read-write lock attribute object created by `pthread_rwlockattr_init()`.

Don’t use a destroyed read-write lock attribute object reinitializing it by calling `pthread_rwlockattr_init()`.

**Returns:**

- **EOK** Success.
- **EINVAL** The object specified by `attr` is invalid.

**Classification:**

POSIX 1003.1 THR
**pthread_rwlockattr_destroy()**

### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`pthread_rwlockattr_getpshared()`, `pthread_rwlockattr_init()`, `pthread_rwlockattr_setpshared()`
Get the process-shared attribute of a read-write lock attribute object

Synopsis:

```c
#include <pthread.h>

int pthread_rwlockattr_getpshared(
    const pthread_rwlockattr_t* attr,
    int* pshared);
```

Arguments:

- `attr` A pointer to the `pthread_rwlockattr_t` object that you want to get the attribute from.
- `pshared` A pointer to a location where the function can store the process-shared attribute.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlockattr_getpshared()` function gets the the process-shared attribute for the read-write lock attribute object specified by `attr`, storing it in `pshared`.

To let any thread with access to the read-write lock object’s memory operate it, the process-shared attribute must be set to `PTHREAD_PROCESS_SHARED`, even if those threads are in different processes. Set the process-shared attribute to `PTHREAD_PROCESS_PRIVATE` to limit access to threads in the current process.

Returns:

- `EOK` Success.
- `EINVAL` The read-write lock attribute object specified by `attr` is invalid.
Classification:

POSIX 1003.1 THR TSH

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_rwlockattr_destroy()`, `pthread_rwlockattr_init()`, `pthread_rwlockattr_setpshared()`
Create a read-write lock attribute object

**Synopsis:**

```c
#include <pthread.h>

int pthread_rwlockattr_init(
    pthread_rwlockattr_t* attr);
```

**Arguments:**

- `attr` A pointer to the `pthread_rwlockattr_t` object that you want to initialize.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_rwlockattr_init()` function initializes the specified read-write lock attribute object to its default values.

Changes made to a read-write lock attribute object changes after it’s been used to initialize a read-write lock won’t affect the previously initialized read-write locks.

**Returns:**

- `EOK` Success.
- `ENOMEM` There isn’t enough memory available to initialize `attr`.

**Classification:**

- POSIX 1003.1 THR
**pthread_rwlockattr_init()**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_rwlockattr_destroy()`, `pthread_rwlockattr_getpshared()`, `pthread_rwlockattr_setpshared()`
Set the process-shared attribute of a read-write lock attribute object

Synopsis:

```c
#include <pthread.h>

int pthread_rwlockattr_setpshared(
    pthread_rwlockattr_t* attr,
    int pshared );
```

Arguments:

- `attr` A pointer to the `pthread_rwlockattr_t` object that you want to set the attribute for.
- `pshared` The new value of the process-shared attribute; one of:
  - `PTHREAD_PROCESS_SHARED` — let any thread with access to the read-write lock object’s memory operate it, even if those threads are in different processes.
  - `PTHREAD_PROCESS_PRIVATE` — limit access to threads in the current process.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pthread_rwlockattr_setpshared()` function sets the process-shared attribute for the read-write lock attribute object specified by `attr` to `pshared`.

Returns:

- `EOK` Success.
- `EINVAL` The `pshared` argument is invalid.
**Classification:**

POSIX 1003.1 THR TSH

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_rwlockattr_destroy()`, `pthread_rwlockattr_getpshared()`, `pthread_rwlockattr_init()`
**pthread_self()**

Get the calling thread's ID

**Synopsis:**

```c
#include <pthread.h>

pthread_t pthread_self( void );
```

**Library:**

```c
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_self()` function returns the thread ID of the calling thread.

**Returns:**

The ID of the calling thread.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_create()`, `pthread_equal()`
**pthread_setcancelstate()**

Set a thread’s cancellation state

**Synopsis:**

```c
#include <pthread.h>

int pthread_setcancelstate( int state,
                          int* oldstate );
```

**Arguments:**

- `state` The new cancellation state.
- `oldstate` A pointer to a location where the function can store the old cancellation state.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_setcancelstate()` function sets the calling thread’s cancellation state to `state` and returns the previous cancellation state in `oldstate`.

The cancellation state can have the following values:

- `PTHREAD_CANCEL_DISABLE`
  
  Cancellation requests are held pending.

- `PTHREAD_CANCEL_ENABLE`

  Cancellation requests may be acted on according to the cancellation type; see `pthread_setcanceltype()`.

The default cancellation state for a thread is `PTHREAD_CANCEL_ENABLE`. 

C Library — P to R 2333
**pthread_setcancelstate()**

You can set this attribute (in a non-POSIX way) before creating the thread; for more information, see “QNX extensions,” in the documentation for `pthread_create()`.

**Returns:**

- **EOK** Success.
- **EINVAL** The cancellation state specified by `state` is invalid.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_cancel()`, `pthread_setcanceltype()`, `pthread_testcancel()`
**pthread_setcanceltype()**

Set a thread's cancellation type

**Synopsis:**

```c
#include <pthread.h>

int pthread_setcanceltype( int type,
                           int* oldtype );
```

**Arguments:**

- `type` The new cancellation type.
- `oldtype` A pointer to a location where the function can store the old cancellation type.

**Library:**

libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `pthread_setcanceltype()` function sets the calling thread’s cancellation type to `type` and returns the previous cancellation type in `oldtype`.

The cancellation type can have the following values:

- **PTHREAD_CANCELASYNCHRONOUS**
  - If cancellation is enabled, new or pending cancellation requests may be acted on immediately.

- **PTHREAD_CANCELDDEFERRED**
  - If cancellation is enabled, cancellation requests are held pending until a cancellation point.

The default cancellation state for a thread is **PTHREAD_CANCELDDEFERRED**. Note that the standard POSIX and C library calls aren’t asynchronous-cancellation safe.
You can set this attribute (in a non-POSIX way) before creating the thread; for more information, see “QNX extensions,” in the documentation for `pthread_create()`.

**Returns:**

- EOK         Success.
- EINVAL      Invalid cancelability type `type`.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_cancel()`, `pthread_setcancelstate()`, `pthread_testcancel()`
**Synopsis:**

```c
#include <pthread.h>

int pthread_setconcurrency( int new_level );
```

**Arguments:**

- `new_level` The new value for the concurrency level.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

QNX Neutrino doesn’t support the multiplexing of user threads on top of several kernel scheduled entities. As such, the `pthread_setconcurrency()` and `pthread_getconcurrency()` functions are provided for source code compatibility but they have no effect when called. To maintain the function semantics, the `new_level` parameter is saved when `pthread_setconcurrency()` is called so that a subsequent call to `pthread_getconcurrency()` returns the same value.

**Returns:**

- `EOK` Success.
- `EINVAL` Negative argument `new_level`.
- `EAGAIN` The value specified by `new_level` would cause a system resource to be exceeded.

**Classification:**

POSIX 1003.1 XSI
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_getconcurrency()`
**Synopsis:**

```c
#include <pthread.h>

int pthread_setname_np(pthread_t tid,
           const char* newname);
```

**Arguments:**

- `tid` The ID of the thread you want to name.
- `newname` NULL, or a NULL-terminated string that specifies the new name.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_setname_np()` function sets the name of the specified thread to `newname`. If `newname` is `NULL`, the function deletes any name already assigned to the thread. The “np” in the function’s name stands for “non-POSIX.”

This function was added in the QNX Neutrino Core OS 6.3.2.

**Returns:**

- `EOK` Success.
- `E2BIG` The name is larger than the accepted size.
- `EINVAL` The name buffer length is invalid or smaller than the new name length.
**Description:**

The process doesn’t have superuser capabilities.

**Classification:**

QNX Neutrino

**Safety:**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_getname_np()`
pthread_setschedparam()

Set thread scheduling parameters

Synopsis:

```c
#include <pthread.h>

int pthread_setschedparam(
    pthread_t thread,
    int policy,
    const struct sched_param *param);
```

Arguments:

- **thread**: The ID of the thread that you want to get the scheduling parameters for. You can get a thread ID by calling `pthread_create()` or `pthread_self()`.

- **policy**: The new scheduling policy; one of:
  - `SCHED_FIFO` — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher priority thread.
  - `SCHED_RR` — the same as `SCHED_FIFO`, except threads at the same priority level timeslice (round robin) every $4 \times$ the clock period (see `ClockPeriod()`).
  - `SCHED_OTHER` — currently the same as `SCHED_RR`.
  - `SCHED_SPORADIC` — sporadic scheduling.

For more information, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

- **param**: A pointer to a `sched_param` structure that specifies the scheduling parameters that you want to use.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `pthread_setschedparam()` function sets the scheduling policy and associated scheduling parameters of thread `thread` to the values specified in `policy` and `param`.

Returns:

- **EOK**: Success.
- **EINVAL**: Invalid scheduling policy `policy` or parameters `param`.
- **ENOTSUP**: Unsupported scheduling policy `policy` or parameters `param`.
- **EPERM**: Insufficient privilege to modify scheduling policy `policy` or parameters `param`.
- **ESRCH**: Invalid thread ID `thread`.

Classification:

- POSIX 1003.1 THR TPS

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_getschedparam()`, `sched_param`


**Synopsis:**
```c
#include <pthread.h>

int pthread_setspecific( pthread_key_t key,
                         const void* value );
```

**Arguments:**
- `key`: The key associated with the data that you want to set. See `pthread_key_create()`.
- `value`: The value that you want to store.

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_setspecific()` function binds the thread specific data value `value` with the thread specific data key `key`.

You can call this function from within a thread-specific data destructor function.

---

You must call this function with a key that you got from `pthread_key_create()`. You can’t use a key after destroying it with `pthread_key_delete()`.

**Returns:**

- **EOK**: Success.
- **ENOMEM**: Insufficient memory to store thread specific data value `value`.
- **EINVAL**: Invalid thread specific data key `key`. 
**Examples:**

See `pthread_key_create()`.  

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

Calling `pthread_setspecific()` with a non-NULL value may result in lost storage or infinite loops unless value was returned by `pthread_key_create()`.

**See also:**

`pthread_key_create()`, `pthread_getspecific()`


**Synopsis:**

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t* set, sigset_t* oset);
```

**Arguments:**

- **how**: How you want to change the signal mask; one of:
  - SIG_BLOCK — make the resulting signal mask the union of the current signal mask and the signal set `set`.
  - SIG_UNBLOCK — make the resulting signal mask the intersection of the current signal mask and the complement of the signal set `set`.
  - SIG_SETMASK — make the resulting signal mask the signal set `set`.

  This argument is valid only if `set` is non-NULL.

- **set**: A pointer to a `sigset_t` object that specifies the signals that you want to affect in the mask.

- **oset**: NULL, or a pointer to a `sigset_t` object where the function can store the thread’s old signal mask.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_sigmask()` function is used to examine and/or change the calling thread’s signal mask. If `set` is non-NULL, the thread’s signal mask is set to `set`. If `oset` is non-NULL, the thread’s old signal mask is returned in `oset`.

You can’t block the SIGKILL and SIGSTOP signals.
**pthread_sigmask()**

Returns:

- EOK: Success.
- EINVVAL: Invalid *how* parameter.

Classification:

- POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- *sigprocmask()*
**pthread_sleepon_broadcast()**

Unblock waiting threads

**Synopsis:**

```c
#include <pthread.h>

int pthread_sleepon_broadcast(
    const volatile void * addr);
```

**Arguments:**

- `addr` The handle that the threads are waiting on. The value of `addr` is typically a data structure that controls a resource.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_sleepon_broadcast()` function unblocks all threads currently waiting on `addr`. The threads are unblocked in priority order.

You should use `pthread_sleepon_broadcast()` or `pthread_sleepon_signal()`, depending on the task you’re doing:

Mapping a single predicate to one address

Use `pthread_sleepon_signal()`.

If you use `pthread_sleepon_broadcast()`, you must recheck the predicate and reblock if necessary. The first thread to wake up owns the lock; all others must go back to sleep.

If you use `pthread_sleepon_signal()`, you don’t have to recheck the predicate. One thread owns the lock at a time.

Mapping multiple predicates to one address

You need to use `pthread_sleepon_broadcast()` to wake up all blocked threads. You must recheck the predicates and reblock if
pthread_sleepon_broadcast() © 2007, QNX Software Systems GmbH & Co.

necessary. You should try to map only one predicate to one address.
Don’t use pthread_sleepon_signal() in this case; it could result in a deadlock.

Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid sleepon address.</td>
</tr>
</tbody>
</table>

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_broadcast(), pthread_sleepon_signal(),
pthread_sleepon_lock(), pthread_sleepon_unlock(),
pthread_sleepon_wait()
Synopsis:

```c
#include <pthread.h>

int pthread_sleepon_lock( void );
```

Library:

```
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.
```

Description:

The `pthread_sleepon_lock()` function calls `pthread_mutex_lock()` on a mutex associated with the `pthread_sleepon*` class of functions. You should call this function before testing conditions that determine whether you need to call `pthread_sleepon_wait()`, `pthread_sleepon_signal()`, or `pthread_sleepon_broadcast()`. This mutex prevents other threads from changing the conditions between the time you examine and act upon them.

This function may be implemented as a simple macro.

Returns:

- `EOK` Success.
- `EDEADLK` The calling thread already owns the controlling mutex.
- `EAGAIN` On the first use of `pthread_sleepon_lock()`, all kernel mutex objects were in use.

Classification:

```
QNX Neutrino
```
**pthread_sleepon_lock()**

© 2007, QNX Software Systems GmbH & Co. KG.

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`pthread_mutex_lock()`, `pthread_sleepon_broadcast()`, `pthread_sleepon_signal()`, `pthread_sleepon_unlock()`, `pthread_sleepon_wait()`
**Synopsis:**

```
#include <pthread.h>

int pthread_sleepon_signal(
    const volatile void * addr);
```

**Arguments:**

- `addr`  The handle that the threads are waiting on. The value of `addr` is typically a data structure that controls a resource.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_sleepon_signal()` function unblocks the highest priority thread waiting on `addr`.

You should use `pthread_sleepon_broadcast()` or `pthread_sleepon_signal()`, depending on the task you’re doing:

Mapping a single predicate to one address

Use `pthread_sleepon_signal()`.

If you use `pthread_sleepon_broadcast()`, you must recheck the predicate and reblock if necessary. The first thread to wake up owns the lock; all others must go back to sleep.

If you use `pthread_sleepon_signal()`, you don’t have to recheck the predicate. One thread owns the lock at a time.

Mapping multiple predicates to one address

You need to use `pthread_sleepon_broadcast()` to wake up all blocked threads. You must recheck the predicates and reblock if
necessary. You should try to map only one predicate to one address.
Don’t use `pthread_sleepon_signal()` in this case; it could result in a deadlock.

Returns:

- `EOK` Success.
- `EINVAL` Invalid sleepon address.

Classification:

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_cond_signal()`, `pthread_sleepon_broadcast()`, `pthread_sleepon_lock()`, `pthread_sleepon_unlock()`, `pthread_sleepon_wait()`
**Synopsis:**

```c
#include <pthread.h>

int pthread_sleepon_timedwait(
    const volatile void * addr,
    uint64_t nsec);
```

**Arguments:**

- `addr`: The handle that you want the thread to wait for. The value of `addr` is typically a data structure that controls a resource.
- `nsec`: A limit on the amount of time to wait, in nanoseconds.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_sleepon_timedwait()` function uses a mutex and a condition variable to sleep on a handle, `addr`.

If `nsec` is nonzero, then `pthread_sleepon_timedwait()` calls `pthread_cond_timedwait()`. If the `pthread_cond_timedwait()` times out, then `pthread_sleepon_timedwait()` returns ETIMEDOUT. If `nsec` is zero, then `pthread_sleepon_timedwait()` calls `pthread_cond_wait()` instead.

The `pthread_sleepon*()` functions provide a simple, uniform way to wait on a variety of resources in a multithreaded application. For example, a multithreaded filesystem may wish to wait on such diverse things as a cache block, a file lock, an operation complete and many others. For example, to wait on a resource:

```c
pthread_sleepon_lock();

while((ptr = cachelist->free) == NULL) {
```
To start an operation and wait upon its completion:

```c
/* Line up for access to the driver */
pthread_sleepon_lock();
if(driver->busy) {
    pthread_sleepon_timedwait(&driver->busy, 1000);
}

/* We now have exclusive use of the driver */
driver->busy = 1;
driver_start(driver); /* This should be relatively fast */

/* Wait for something to signal driver complete */
pthread_sleepon_timedwait(&driver->complete, 1000);
pthread_sleepon_unlock();

/* Get the status/data */
driver_complete(driver);

/* Release control of the driver and signal anyone waiting */
pthread_sleepon_lock();
driver->busy = 0;
pthread_sleepon_signal(&driver->busy);
pthread_sleepon_unlock();
```

The use of a `while` loop instead of an `if` handles the case where the wait on `addr` is woken up using `pthread_sleepon_broadcast()`.

You must call `pthread_sleepon_lock()`, which acquires the controlling mutex for the condition variable and ensures that another thread won’t enter the critical section between the test, block and use of the resource. Since `pthread_sleepon_timedwait()` calls `pthread_cond_timedwait()`, it releases the controlling mutex when it blocks. It reacquires the mutex before waking up.

The wakeup is accomplished by another thread’s calling `pthread_sleepon_signal()`, which wakes up a single thread, or `pthread_sleepon_broadcast()`, which wakes up all threads blocked on `addr`. Threads are woken up in priority order. If there’s more than one thread with the same highest priority, the one that has been waiting the longest is woken first.
A single mutex and one condition variable for each unique address that’s currently being blocked on are used. The total number of condition variables is therefore equal to the number of unique \textit{addr}s that have a thread waiting on them. This also means that the maximum number of condition variables never exceeds the number of threads. To accomplish this, condition variables are dynamically created as needed and placed upon an internal freelist for reuse when not.

You might find the \texttt{pthread\_sleepon\_\*()} functions easier to use and understand than condition variables. They also resemble the traditional \texttt{sleepon()} and \texttt{wakeup()} functions found in Unix kernels. They can be implemented as follows:

\begin{verbatim}
int _sleepon(void *addr) {
    int ret;
    if((ret = pthread_sleepon_lock()) == EOK) {
        ret = pthread_sleepon_timedwait(addr, 1000);
        pthread_sleepon_unlock();
    }
    return ret;
}

void _wakeup(void *addr) {
    if(pthread_sleepon_lock() == EOK) {
        pthread_sleepon_broadcast(addr, 1000);
        pthread_sleepon_unlock();
    }
}
\end{verbatim}

Note that in most Unix kernels, a thread runs until it blocks and thus need not worry about protecting the condition it checks with a mutex. Likewise when a Unix \texttt{wakeup()} is called, there isn’t an immediate thread switch. Therefore, you can use only the above simple routines (_\texttt{wakeup()} and _\texttt{sleepon()} if all your threads run with SCHED\_FIFO scheduling and at the same priority, thus more closely mimicking Unix kernel scheduling.
Returns:

- `EDEADLK`: The calling thread already owns the controlling mutex.
- `ETIMEDOUT`: The time specified by `nsec` has passed.
- `EOK`: Success.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `pthread_cond_wait()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_sleepon_broadcast()`, `pthread_sleepon_lock()`, `pthread_sleepon_signal()`, `pthread_sleepon_unlock()`, `pthread_sleepon_wait()`, `sched_setscheduler()`
**pthread_sleepon_unlock()**

Unlock the pthread_sleepon*() functions

**Synopsis:**

```c
#include <pthread.h>

int pthread_sleepon_unlock( void );
```

**Library:**

```
libc
```

Use the \-l c option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `pthread_sleepon_unlock()` function calls `pthread_mutex_unlock()` on a mutex associated with the `pthread_sleepon*()` class of functions. You should call it at the end of a critical section entered by `pthread_sleepon_lock()`.

This function may be implemented as a simple macro.

**Returns:**

- **EOK** Success.
- **EPERM** The current thread doesn’t own the controlling mutex.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

- `pthread_mutex_unlock()`, `pthread_sleepon_broadcast()`,
- `pthread_sleepon_lock()`, `pthread_sleepon_signal()`,
- `pthread_sleepon_wait()`
**pthread_sleepon_wait()**

Make a thread sleep while waiting

**Synopsis:**

```c
#include <pthread.h>

int pthread_sleepon_wait( const volatile void * addr );
```

**Arguments:**

`addr` The handle that you want the thread to wait for. The value of `addr` is typically a data structure that controls a resource.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_sleepon_wait()` function uses a mutex and a condition variable to sleep on a handle, `addr`.

The `pthread_sleepon*` functions provide a simple, uniform way to wait on a variety of resources in a multithreaded application. For example, a multithreaded filesystem may wish to wait on such diverse things as a cache block, a file lock, an operation complete and many others. For example, to wait on a resource:

```c
pthread_sleepon_lock();
while((ptr = cachelist->free) == NULL) {
    pthread_sleepon_wait(cachelist);
}
```

```c
cachelist->free = ptr->free;
```

```c
pthread_sleepon_unlock();
```

To start an operation and wait for its completion:

```c
/* Line up for access to the driver */
pthread_sleepon_lock();
if(driver->busy) {
    pthread_sleepon_wait(&driver->busy);
```
/* We now have exclusive use of the driver */
driver->busy = 1;
driver_start(driver); /* This should be relatively fast */

/* Wait for something to signal driver complete */
pthread_sleepon_wait(&driver->complete);
pthread_sleepon_unlock();

/* Get the status/data */
driver_complete(driver);

/* Release control of the driver and signal anyone waiting */
pthread_sleepon_lock();
driver->busy = 0;
pthread_sleepon_signal(&driver->busy);
pthread_sleepon_unlock();

pthread_exit(NULL);

Choose carefully when you decide whether to use a while loop

- If the wait on addr is woken up using
  pthread_sleepon_broadcast(), you must use a while loop.

- If threads are woken up using pthread_sleepon_signal(), you may
  use the if conditional if the design of the program guarantees
  proper synchronization and scheduling among contending threads.
  This is guaranteed in the above example, assuming that none of the
  threads attempt to reacquire the driver resource (i.e. pthread_exit() 
call).

If you’re in doubt, use a while loop, because it guarantees access to 
the desired resource.

You must call pthread_sleepon_lock(), which acquires the controlling
mutex for the condition variable and ensures that another thread won’t
enter the critical section between the test, block and use of the
resource. Since pthread_sleepon_wait() calls pthread_cond_wait(),
it releases the controlling mutex when it blocks. It reacquires the
mutex before waking up.

The wakeup is accomplished by another thread’s calling
pthread_sleepon_signal(), which wakes up a single thread, or
pthread_sleepon_broadcast(), which wakes up all threads blocked on addr. Threads are woken up in priority order. If there’s more than one thread with the same highest priority, the one that has been waiting the longest is woken first.

A single mutex and one condition variable for each unique address that’s currently being blocked on are used. The total number of condition variables is therefore equal to the number of unique addrs that have a thread waiting on them. This also means that the maximum number of condition variables never exceeds the number of threads. To accomplish this, condition variables are dynamically created as needed and placed upon an internal freelist for reuse when not.

You might find the pthread_sleepon_*( ) functions easier to use and understand than condition variables. They also resemble the traditional sleepon() and wakeup() functions found in Unix kernels. They can be implemented as follows:

```c
int _sleepon(void *addr) {
    int ret;
    if((ret = pthread_sleepon_lock()) == EOK) {
        ret = pthread_sleepon_wait(addr);
        pthread_sleepon_unlock();
    }
    return ret;
}

void _wakeup(void *addr) {
    if(pthread_sleepon_lock() == EOK) {
        pthread_sleepon_broadcast(addr);
        pthread_sleepon_unlock();
    }
}
```

Note that in most Unix kernels, a thread runs until it blocks, and thus need not worry about protecting the condition it checks with a mutex. Likewise, when a Unix wakeup() is called, there isn’t an immediate thread switch. Therefore, you can use only the above simple routines (_wakeup() and _sleepon()) if all your threads run with SCHED_FIFO scheduling and at the same priority, thus more closely mimicking Unix kernel scheduling.
pthread_sleepon_wait()

Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EDEADLK</td>
<td>The calling thread already owns the controlling mutex.</td>
</tr>
</tbody>
</table>

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_wait(), pthread_mutex_lock(), pthread_mutex_unlock(), pthread_sleepon_broadcast(), pthread_sleepon_lock(), pthread_sleepon_signal(), pthread_sleepon_unlock(), sched_setscheduler()
**pthread_spin_destroy()**

Destroy a thread spinlock

**Synopsis:**

```c
#include <pthread.h>

int pthread_spin_destroy(
    pthread_spinlock_t * spinner );
```

**Arguments:**

- `spinner` A pointer to the `pthread_spinlock_t` object that you want to destroy.

**Library:**

- libc

Use the `-1 c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_spin_destroy()` function destroys the thread spinlock `spinner`, releasing its resources.

Once you’ve destroyed the spinlock, don’t use it again until you’ve reinitialized it by calling `pthread_spin_init()`.

Calling `pthread_spin_destroy()` gives undefined results when a thread has `spinner` locked or when `spinner` isn’t initialized.

**Returns:**

- **EOK** Success.
- **EBUSY** The thread spinlock `spinner` is in use by another thread and can’t be destroyed.
- **EINVAL** Invalid `pthread_spinlock_t` object `spinner`. 
**Classification:**

POSIX 1003.1 THR SPI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_spin_init()`, `pthread_spin_lock()`, `pthread_spin_trylock()`, `pthread_spin_unlock()`
**Synopsis:**

```c
#include <pthread.h>

int pthread_spin_init( pthread_spinlock_t * spinner,
                      int pshared )
```

**Arguments:**

- `spinner` A pointer to the `pthread_spinlock_t` object that you want to initialize.
- `pshared` The value that you want to use for the process-shared attribute of the spinlock. The possible values are:
  - PTHREAD_PROCESS_SHARED — the spinlock may be operated on by any thread that has access to the memory where the spinlock is allocated, even if it’s allocated in memory that’s shared by multiple processes.
  - PTHREAD_PROCESS_PRIVATE — the spinlock can be operated on only by threads created within the same process as the thread that initialized the spinlock. If threads of differing processes attempt to operate on such a spinlock, the behavior is undefined.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_spin_init()` function allocates the resources required for the thread spinlock `spinner`, and initializes `spinner` to an unlocked state.

Any thread that can access the memory where `spinner` is allocated can operate on the spinlock.
Results are undefined if you call `pthread_spin_init()` on a `spinner` that’s already initialized, or if you try to use a spinlock that hasn’t been initialized.

**Returns:**

Zero on success, or an error number to indicate the error.

**Errors:**

- **EAGAIN** The system doesn’t have the resources required to initialize a new spinlock.
- **EBUSY** The process spinlock, `spinner`, is in use by another thread and can’t be initialized.
- **EINVAL** Invalid `pthread_spinlock_t` object `spinner`.
- **ENOMEM** The system doesn’t have enough free memory to create the new spinlock.

**Classification:**

POSIX 1003.1 THR SPI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_spin_destroy()`, `pthread_spin_lock()`, `pthread_spin_trylock()`, `pthread_spin_unlock()`
**pthread_spin_lock()**

Lock a thread spinlock

**Synopsis:**

```c
#include <pthread.h>

int pthread_spin_lock( pthread_spinlock_t * spinner );
```

**Arguments:**

- `spinner` A pointer to the `pthread_spinlock_t` object that you want to lock.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_spin_lock()` function locks the thread spinlock specified by `spinner`. If `spinner` isn’t immediately available, `pthread_spin_lock()` blocks until `spinner` can be locked.

If a thread attempts to lock a spinlock that’s already locked via `pthread_spin_lock()` or `pthread_spin_trylock()`, the thread returns `EDEADLK`.

**Returns:**

- **EOK** Success.
- **EAGAIN** Insufficient resources available to lock `spinner`.
- **EDEADLK** The calling thread already holds `spinners` lock.
- **EINVAL** Invalid `pthread_spinlock_t` object `spinner`. 
**p**thread**s**pin**_**lock()**

**Classification:**

POSIX 1003.1 THR SPI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

You may not get the desired behavior from this function because the current implementation is through mutexes. In the uncontested case, this gives the same behavior as spinlocks. In the contested case, this function makes a kernel call.

**See also:**

p**thread**_spin_destroy(), p**thread**_spin_init(), p**thread**_spin_trylock(), p**thread**_spin_unlock()
**pthread_spin_trylock()**

Try to lock a thread spinlock

**Synopsis:**

```c
#include <pthread.h>

int pthread_spin_trylock(
    pthread_spinlock_t * spinner );
```

**Arguments:**

- `spinner` A pointer to the `pthread_spinlock_t` object that you want to try to lock.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_spin_trylock()` function attempts to lock the thread spinlock specified by `spinner`. It returns immediately if `spinner` can’t be locked.

If a thread attempts to lock a spinlock that it’s already locked via `pthread_spin_lock()` or `pthread_spin_trylock()`, the thread deadlocks.

**Returns:**

- `EOK` Success.
- `EAGAIN` Insufficient resources available to lock `spinner`.
- `EBUSY` The thread spinlock `spinner` is already locked by another thread.
- `EINVAL` Invalid `pthread_spinlock_t` object `spinner`. 
**Classification:**

POSIX 1003.1 THR SPI

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_spin_destroy()`, `pthread_spin_init()`, `pthread_spin_lock()`, `pthread_spin_unlock()`
**pthread_spin_unlock()**

Unlock a thread spinlock

**Synopsis:**

```c
#include <pthread.h>

int pthread_spin_unlock( spinlock_t * spinner );
```

**Arguments:**

*spinner*  
A pointer to the `pthread_spinlock_t` object that you want to unlock.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_spin_unlock()` function unlocks the thread spinlock specified by *spinner*, which was locked with `pthread_spin_lock()` or `pthread_spin_trylock()`.

If there are threads spinning on *spinner*, the spinlock becomes available, and an unspecified thread acquires the lock.

**Returns:**

- **EOK**  
  Success.
- **EINVAL**  
  Invalid process spinlock *spinner*.
- **EPERM**  
  The calling thread doesn’t hold the lock.

**Classification:**

POSIX 1003.1 THR SPI
**pthread_spin_unlock()**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_spin_destroy()`, `pthread_spin_init()`, `pthread_spin_lock()`, `pthread_spin_trylock()`
**Synopsis:**

```c
#include <pthread.h>

void pthread_testcancel( void );
```

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_testcancel()` function creates a cancellation point in the calling thread. This function has no effect if cancellation is disabled.

**Classification:**

POSIX 1003.1 THR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pthread_cancel()`, `pthread_setcancelstate()`, `pthread_setcanceltype()`, `ThreadCancel()`
**pthread_timedjoin()**

Join a thread, with a time limit

**Synopsis:**

```c
#include <pthread.h>

int pthread_timedjoin(
    pthread_t thread,
    void** value_ptr,
    const struct timespec* abstime);
```

**Arguments:**

- **thread**: The target thread whose termination you’re waiting for.
- **value_ptr**: NULL, or a pointer to a location where the function can store the value passed to `pthread_exit()` by the target thread.
- **abstime**: A pointer to a `timespec` structure that specifies the maximum time to wait for the join, expressed as an absolute time.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `pthread_timedjoin()` function is similar to `pthread_join()`, except that an error of ETIMEDOUT is returned if the join doesn’t occur before the absolute time specified by `abstime` passes (i.e. the system time is greater than or equal to `abstime`):

> If you are not too long, I will wait here for you all my life.
> — Oscar Wilde, *The Importance of Being Earnest*

The `pthread_timedjoin()` function blocks the calling thread until the target thread `thread` terminates, unless `thread` has already terminated.
If `value_ptr` is non-NULL and `pthread_timedjoin()` returns successfully, then the value passed to `pthread_exit()` by the target thread is placed in `value_ptr`. If the target thread has been canceled then `value_ptr` is set to PTHREAD_CANCELED.

The target thread must be joinable. Multiple `pthread_join()`, `pthread_timedjoin()`, `ThreadJoin()`, and `ThreadJoin_r()` calls on the same target thread aren’t allowed. When `pthread_timedjoin()` returns successfully, the target thread has been terminated.

**Returns:**

- **EOK**  
  Success.
- **EBUSY**  
  The thread `thread` is already being joined.
- **EDEADLK**  
  The thread `thread` is the calling thread.
- **EFAULT**  
  A fault occurred trying to access the buffers provided.
- **EINVAL**  
  The thread `thread` isn’t joinable.
- **ESRCH**  
  The thread `thread` doesn’t exist.
- **ETIMEDOUT**  
  The absolute time specified in `abstime` passed before the join occurred.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

pthread_create(), pthread_detach(), pthread_exit(), pthread_join(),
ThreadJoin(), ThreadJoin_r(), timespec
Synopsis:

```
#include <sys/neutrino.h>

struct _pulse {
    uint16_t type;
    uint16_t subtype;
    int8_t code;
    uint8_t zero[3];
    union sigval value;
    int32_t scoid;
};
```

Description:

The _pulse structure describes a pulse, a fixed-size, nonblocking message that carries a small payload (four bytes of data plus a single byte code). The members include:

- **type** _PULSE_TYPE (0)
- **subtype** _PULSE_SUBTYPE (0)
- **code**

  A code that identifies the type of pulse. The QNX Neutrino OS reserves the negative codes, including the following:
  
  - _PULSE_CODE_UNBLOCK
  - _PULSE_CODE_DISCONNECT
  - _PULSE_CODE_THREADDEATH
  - _PULSE_CODE_COIDDEATH
  - _PULSE_CODE_NET_ACK,
    _PULSE_CODE_NET_UNBLOCK, and
  - _PULSE_CODE_NET_DETACH — reserved for the io_net resource manager.

You can define your own pulses, with a code in the range from _PULSE_CODE_MINA V AIL through _PULSE_CODE_MAXA V AIL.
value  Information that’s relevant to the code:
   • _PULSE_CODE_UNBLOCK — the receive ID (rcvid) associated with the blocking message.
   • _PULSE_CODE_DISCONNECT — no value defined.
   • _PULSE_CODE_THREADDEATH — the thread ID of the thread that died.
   • _PULSE_CODE_COIDDEATH — the connection ID of a connection that was attached to a destroyed channel.

For more details, see ChannelCreate().
If you define your own pulses, you can decide what information you want to store in this field.

scoid  Server connection ID.

Classification:
QNX Neutrino

See also:
ChannelCreate(), MsgReceive(), MsgReceivePulse(),
MsgReceivePulsev(), MsgReceivev(), MsgSendPulse(), sigevent
**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int pulse_attach( dispatch_t * dpp,
        int flags,
        int code,
        int (*func)( message_context_t * ctp,
                      int code,
                      unsigned flags,
                      void * handle ),
        void * handle );
```

**Arguments:**

- **dpp** The dispatch handle, as returned by `dispatch_create()`.
- **flags** Currently, the following flag is defined in `<sys/dispatch.h>`:
  - MSG_FLAG_ALLOC_PULSE — allocate and attach a pulse code that’s different than any other code that was either given to `pulse_attach()` through the `code` argument, or allocated by `pulse_attach()`. The allocated code is in the range _PULSE_CODE_MINA V AIL and _PULSE_CODE_MAXA V AIL.
- **code** The pulse code that you want to attach the function to.
- **func** The function that you want to call when a message in the given range is received; see “Handler function,” in the documentation for `message_attach()`.
- **handle** An arbitrary handle that you want to associate with data for the defined message range. This handle is passed to `func`. 

**pulse_attach()**

Attach a handler function to a pulse code
Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pulse_attach()` function attaches a pulse `code` to a user-supplied function `func`. You can use the same function `func` with `message_attach()`.

When the resource manager receives a pulse that matches `code`, it calls `func`. This user-supplied function is responsible for doing any specific work needed to handle the pulse pointed to by `ctp->msg.pulse`. The `handle` passed to the function is the `handle` initially passed to `pulse_attach()`. The `handle` may be a device entry you want associated with the pulse `code`.

You typically use `pulse_attach()` to associate pulses generated by interrupt handlers or timers with a routine in the main program of your resource manager. By examining `ctp->rcvid`, the `func` function can determine whether a pulse or message was received.

Returns:

If `MSG_FLAG_ALLOC_PULSE` is specified, the function returns the allocated pulse code; otherwise, it returns the `code` that’s passed in. On failure, -1 is returned (`errno` is set).

Errors:

- `EAGAIN` Couldn’t allocate a pulse `code`.
- `EINVAL` The pulse `code` is out of range, or it’s already registered.
- `ENOMEM` Insufficient memory to allocate internal data structures.
Examples:

```c
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int my_func(...)
{
    ;
}

int main( int argc, char **argv )
{
    dispatch_t *dpp;
    int flag = 0, code, mycode;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
    ;

    mycode = ...;

    if ( (code = pulse_attach( dpp, flag, mycode, &my_func, NULL)) == -1 ) {
        fprintf( stderr, "Failed to attach code %d.\n", mycode );
        return 1;
    }
    /* else successfully attached a pulse code */
    ;
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

Classification:

QNX Neutrino
### pulse_attach()

#### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### See also:

- `message_attach()`, `pulse_detach()`
- “Components of a Resource Manager” section of the Writing a Resource Manager chapter in the *Programmer’s Guide*.

© 2007, QNX Software Systems GmbH & Co. KG.
Detach a handler function from a pulse code

Synopsis:
```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int pulse_detach( dispatch_t * dpp,
                 int code,
                 int flags);
```

Arguments:
- `dpp` The dispatch handle, as returned by `dispatch_create()`.
- `code` The pulse code that you want to detach.
- `flags` Reserved; pass 0 for this argument.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `pulse_detach()` function detaches the pulse `code`, for dispatch handle `dpp`, that was attached with `pulse_attach()`.

Returns:
- 0 Success.
- -1 The pulse `code` doesn’t match any attached pulse code.

Examples:
```
#include <sys/dispatch.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int my_func( ... ) {
```
int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int flag=0, code, mycode;

    if ( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \n dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    if ( (code = pulse_attach( dpp, flag, mycode, &my_func, NULL)) == -1 ) {
        fprintf ( stderr, "Failed to attach pulse code %d.\n", mycode );
        return 1;
    }

    if ( pulse_detach ( dpp, code, flag ) == -1 ) {
        fprintf ( stderr, "Failed to detach code %d.\n", code );
        return 1;
    }
    /* else message was detached */

    For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

Classification:

QNX Neutrino

Safety

Cancellation point  No

    continued...
**pulse_detach()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`message_detach()`, `pulse_attach()`

“Components of a Resource Manager” section of the Writing a Resource Manager chapter in the *Programmer’s Guide*. 
Synopsis:

```
#include <stdio.h>

int putc( int c,
          FILE* fp );
```

Arguments:

- `c` The character that you want to write.
- `fp` The stream you want to write the character on.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putc()` macro writes the character `c`, cast as `(int)(unsigned char)`, to the output stream designated by `fp`.

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurs (`errno` is set).

Examples:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE* fp;
    int c;

    fp = fopen( "file", "r" );
    if( fp != NULL ) {
        while( (c = fgetc( fp )) != EOF ) {
            putc( c, stdout );
        }
    }
}
```
fclose( fp );
}

return EXIT_SUCCESS;
}

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

putc() is a macro.

See also:

erno, ferror(), fopen(), fprintf(), fputchar(), fputs(), getc(),
getc_unlocked(), getchar(), getchar_unlocked(), putchar(),
putchar_unlocked(), putc_unlocked(), puts()
**putc_unlocked()**

Write a character to an output stream

**Synopsis:**
```
#include <stdio.h>

int putc_unlocked( int c,    
        FILE *stream );
```

**Arguments:**
- `c` The character that you want to write.
- `stream` The stream you want to write the character on.

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `putc_unlocked()` function is a thread-unsafe version of `putc()`. You can use it safely only when the invoking thread has locked `stream` using `flockfile()` (or `ftrylockfile()`) and `funlockfile()`.

**Returns:**
The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred (`errno` is set).

**Classification:**
POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
putc_unlocked()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

getc(), getchar(), getchar_unlocked(), getc_unlocked(), putc(), putchar(), putchar_unlocked()
**Synopsis:**

```c
#include <stdio.h>

int putchar( int c );
```

**Arguments:**

- `c`  
  The character that you want to write.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `putchar()` function writes the character `c`, cast as `(int)(unsigned char)`, to the `stdout` stream. It’s equivalent to:

```c
fputc( c, stdout );
```

**Returns:**

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurs (`errno` is set).

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    int c;

    fp = fopen( "file", "r" );
    c = fgetc( fp );
    while( c != EOF ) {
        putchar( c );
        c = fgetc( fp );
    }
}```
}  
  fclose( fp );  
  
  return EXIT_SUCCESS;  
}  

Classification:  
ANSI, POSIX 1003.1  

Safety  

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:  
errno, fputc(), fputchar(), fputs(), getc(), getc_unlocked(), getchar(),  
getchar_unlocked(),putc(), putchar_unlocked(), putc_unlocked()
putchar_unlocked()
Write a character to stdout

Synopsis:

```c
#include <stdio.h>

int putchar_unlocked( int c );
```

Arguments:

- `c` The character that you want to write.

Library:

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putchar_unlocked()` function is a thread-unsafe version of `putchar()`. You can use it safely only when the invoking thread has locked `stdout` using `flockfile()` (or `ftrylockfile()`) and `funlockfile()`.

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred (`errno` is set).

Classification:

- POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

`getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, `putc()`, `putc_unlocked()`, `putchar()`
**Synopsis:**

```c
#include <stdlib.h>

int putenv( const char *env_name );
```

**Arguments:**

- `env_name` The name of the environment, and what you want to do to it; see below.

**Library:**

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `putenv()` function uses `env_name`, in the form `name=value`, to set the environment variable `name` to `value`. This function alters `name` if it exists, or creates a new environment variable.

In either case, `env_name` becomes part of the environment; subsequent modifications to the string pointed to by `env_name` affect the environment.

The space for environment names and their values is limited. Consequently, `putenv()` can fail when there’s insufficient space remaining to store an additional value.

If `env_name` isn’t a literal string, you should duplicate the string, since `putenv()` doesn’t copy the value. For example:

```c
putenv( strdup( buffer ) );
```
Returns:

0    Success.
-1   An error occurred; errno is set.

Errors:

ENOMEM There wasn’t enough memory to expand the environment.

Examples:

The following gets the string currently assigned to INCLUDE and displays it, assigns a new value to it, gets and displays it, and then removes INCLUDE from the environment.

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *path;
    path = getenv( "INCLUDE" );
    if( path != NULL ) {
        printf( "INCLUDE=%s\n", path );
    }

    if( putenv( "INCLUDE=/src/include" ) != 0 ) {
        printf( "putenv() failed setting INCLUDE\n" );
        return EXIT_FAILURE;
    }

    path = getenv( "INCLUDE" );
    if( path != NULL ) {
        printf( "INCLUDE=%s\n", path );
    }

    unsetenv( "INCLUDE" );

    return EXIT_SUCCESS;
}
```

This program produces the following output:

```
INCLUDE=/usr/nto/include
INCLUDE=/src/include
```

September 10, 2007 C Library — P to R 2395
Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

Never use `putenv()` with an automatic variable.

The `putenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

`clearenv()`, `environ`, `errno`, `getenv()`, `setenv()`, `unsetenv()`
Synopsis:

```c
#include <stdio.h>

int puts( const char *buf );
```

Arguments:

- `buf` A pointer to the zero-terminated string that you want to write.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `puts()` function writes the character string pointed to by `buf` to the `stdout` stream, and appends a newline character to the output. The terminating NUL character of `buf` isn’t written.

Returns:

A nonnegative value for success, or EOF if an error occurs (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;
    char buffer[80];

    fp = freopen( "file", "r", stdin );
    while( gets( buffer ) != NULL ) {
        puts( buffer );
    }
    fclose( fp );

    return EXIT_SUCCESS;
}
```
puts()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fputs(), gets(), putc()
putspent()

Put an entry into the shadow password database

Synopsis:

```c
#include <sys/types.h>
#include <shadow.h>

int putspent( const struct spwd* p,
              FILE* fp );
```

Arguments:

- `p` A pointer to a `spwd` structure that contains the entry that you want to write.
- `fp` The stream that you want to write the entry on.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putspent()` function writes a shadow password entry into the specified file. This function is the inverse of `getspent()`.

Given a pointer to a `spwd` structure created by the `getspent()` or the `getspnam()` routine, `putspent()` writes a line on the stream `fp`, which matches the format of `</etc/shadow>`. The `spwd` structure contains the following members:

```c
char *sp_namp; /* name */
char *sp_pwdp; /* encrypted password */
long sp_changes; /* last changed */
long sp_max; /* days (min) to change */
long sp_min; /* days (max) to change */
long sp_warn; /* days to warn */
long sp_inact; /* days of inactivity */
long sp_expire; /* date to auto-expire */
long sp_flag; /* reserved */
```

If the `sp_min`, `sp_max`, `sp_changes`, `sp_warn`, `sp_inact`, or `sp_expire` field of the structure is -1, or if `sp_flag = 0`, the corresponding `</etc/shadow>` field is cleared.
Returns:

Zero.

Errors:

The `putspent()` function uses the following functions, and as a result `errno` can be set to an error for any of these calls:

- `fclose()`
- `fgets()`
- `fopen()`
- `fseek()`
- `fseek()`
- `rewind()`

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <shadow.h>

/*
 * This program adds a user and password to
 * a temporary file which can then be used with
 * `fgetspent()` (of course the password
 * string should be encrypted already etc.)
 */

int main( int argc, char** argv )
{
    FILE* fp;
    struct spwd  sp;
    char pwbuf[80], nambuf[80];

    memset(&sp, 0, sizeof(sp));
    if (argc < 2) {
        printf("%s filename \n", argv[0]);
        return(EXIT_FAILURE);
    }

    if (!fp = fopen(argv[1], "w")) {
        fprintf(stderr, "Can’t open file %s \n", argv[1]);
        perror("Problem ");
    }
```
putspent()

```c
return(1);
}

printf("Enter a userid: ");
if (!gets(nambuf)) {
    fprintf(stderr, "Can’t get username string\n");
}
sp.sp_namp = nambuf;

printf("Enter a password: ");
if (!gets(pwbuf)) {
    fprintf(stderr, "Can’t get username password\n");
}
sp.sp_pwdp = pwbuf;

putspent(&sp, fp);
fclose(fp);
return(EXIT_SUCCESS);
}
```

Classification:
Unix

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`errno, getgrent(), getlogin(), getpwnam(), getpwuid(), getspent(),
getspnam(), setspent()`
**pututline()**

Write an entry in the user-information file

**Synopsis:**
```c
#include <utmp.h>

void pututline( struct utmp * __utmp );
```

**Arguments:**

__utmp A pointer to the utmp structure for the entry that you want to add.

**Library:**

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).

**Description:**

The pututline() function writes out the supplied utmp structure into the utmp file.

It uses getutid() to search forward for the proper place if it finds that it isn’t already there. Normally, you should search for the proper entry by calling getutent(), getutid(), or getutline(). If so, pututline() doesn’t search. If pututline() doesn’t find a matching slot for the new entry, it adds a new entry to the end of the file.

When called by a non-root user, pututline() invokes a setuid() root program to verify and write the entry, since the file specified in _PATH_UTMP is normally writable only by root. In this event, the ut_name field must correspond to the actual user name associated with the process; the ut_type field must be either USER_PROCESS or DEAD_PROCESS; the ut_line field must be a device-special file and be writable by the user.
Returns:

A pointer to the `utmp` structure.

Files:

`_PATH_UTMP`

Specifies the user information file.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The most current entry is saved in a static structure. Copy it before making further accesses.

On each call to either `getutid()` or `getutline()`, the routine examines the static structure before performing more I/O. If the contents of the static structure match what it’s searching for, the function looks no further. For this reason, to use `getutline()` to search for multiple occurrences, zero out the static area after each success, or `getutline()` will return the same structure over and over again.

There’s one exception to the rule about emptying the structure before further reads are done: the implicit read done by `pututline()` (if it finds that it isn’t already at the correct place in the file) doesn’t hurt the contents of the static structure returned by the `getutent()`, `getutid()` or `getutline()` routines, if the user has just modified those contents and passed the pointer back to `pututline()`.
These routines use buffered standard I/O for input, but `pututline()` uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the `utmp` and `wtmp` files.

See also:

- `endutent()`, `getutent()`, `getutid()`, `getutline()`, `setutent()`, `utmp`, `utmpname()`
- `login` in the *Utilities Reference*
**Synopsis:**

```c
#include <wchar.h>

int putw( int w, FILE *stream );
```

**Arguments:**

- `w` The word that you want to write.
- `stream` The stream that you want to write a word on.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `putw()` function writes the C `int` (word) `w` to the standard I/O output `stream` (at the position of the file pointer, if defined). The size of a word is the size of an integer, and varies from machine to machine. The `putw()` function neither assumes nor causes special alignment in the file.

**Returns:**

- `0` Success.
- `1` An error occurred; `errno` is set.
Errors:

EFBIG   The file is a regular file and an attempt was made to write at or beyond the offset maximum.

Classification:

Legacy Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Because of possible differences in word length and byte ordering, files written using putw() are machine-dependent, and might not be read correctly using getw() on a different processor.

See also:

errno, ferror(), fopen(), fputc(), fprintf(), fputs(), getw(), putchar(), putchar_unlocked(), putc_unlocked(), puts()
Synopsis:

```c
#include <wchar.h>

wint_t putwc( wchar_t wc,
             FILE * fp );
```

Arguments:

- `wc` The wide character that you want to write.
- `fp` The stream that you want to write the wide character on.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putwc()` functions writes the wide character specified by `wc`, cast as `(wint_t) (wchar_t)`, to the output stream specified by `fp`.

Returns:

The wide character written, cast as `(wint_t) (wchar_t)`, or `WEOF` if an error occurs (`errno` is set).

If `wc` exceeds the valid wide-character range, the value returned is the wide character written, not `wc`.

Errors:

- **EAGAIN** The O_NONBLOCK flag is set for `fp` and would have been blocked by this operation.
- **EBADF** The file descriptor for `fp` isn’t valid for writing.
**putwc()**

EFBIG The file exceeds the maximum file size, the process’s file size limit, or the function can’t write at or beyond the offset maximum.

EINTR A signal terminated the write operation; no data was transferred.

EIO A physical I/O error has occurred or all of the following conditions were met:
- The process is in the background.
- TOSTOP is set.
- The process is blocking/ignoring SIGTTOU.
- The process group is orphaned.

EPIPE Can’t write to pipe or FIFO because it’s closed; a SIGPIPE signal is also sent to the thread.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno, getwc(), getwchar()`

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```
#include <wchar.h>

wint_t putwchar( wchar_t wc );
```

Arguments:

- `wc` The wide character that you want to write.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `putwchar()` function writes the wide character `wc`, cast as `(wint_t)(wchar_t)`, to the `stdout` stream. It’s equivalent to:

```
fputwc( wc, stdout );
```

Returns:

The wide character written, cast as `(wint_t)(wchar_t)` or WEOF if an error occurs (`errno` is set).

- **Errors:**
  - `EAGAIN` The O_NONBLOCK flag is set for `fp` and would have been blocked by this operation.
  - `EBADF` The file descriptor for `fp` isn’t valid for writing.
putwchar()

EFBIG The file exceeds the maximum file size, the process’s file size limit, or the function can’t write at or beyond the offset maximum.

EINTR A signal terminated the write operation; no data was transferred.

EIO A physical I/O error has occurred or all of the following conditions were met:
- The process is in the background.
- TOSTOP is set.
- The process is blocking/ignoring SIGTTOU.
- The process group is orphaned.

EPIPE Can’t write to pipe or FIFO because it’s closed; a SIGPIPE signal is also sent to the thread.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

erno, getwc(), getwchar()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```c
#include <unistd.h>

ssize_t pwrite( int filedes,
                const void* buff,
                size_t nbytes,
                off_t offset );

ssize_t pwrite64( int filedes,
                  const void* buff,
                  size_t nbytes,
                  off64_t offset );
```

Arguments:

- `filedes` The file descriptor for the file you want to write in.
- `buff` A pointer to a buffer that contains the data you want to write.
- `nbytes` The number of bytes to write.
- `offset` The desired position inside the file.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `pwrite()` function performs the same action as `write()`, except that it writes into a given position without changing the file pointer.

The `pwrite64()` function is a 64-bit version of `pwrite()`.
pwrite(), pwrite64()

Returns:

The number of bytes actually written, or -1 if an error occurred (errno is set).

Errors:

EAGAIN The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF The file descriptor, fildes, isn’t a valid file descriptor open for writing.

EFTBG File is too big.

EINTR The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.

EIO A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSPC There’s no free space remaining on the device containing the file.

ENOSYS The pwrite() function isn’t implemented for the filesystem specified by fildes.

EPIPE An attempt was made to write to a pipe (or FIFO) that isn’t open for reading by any process. A SIGPIPE signal is also sent to the process.

Classification:

pwrite() is POSIX 1003.1 XSI; pwrite64() is Large-file support

Safety

Cancellation point Yes

continued...
Safet

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

close(), creat(), dup(), dup2(), errno, fcntl(), lseek(), open(), pipe(), pread(), read(), readv(), select(), write(), writev()
Encrypt a password (QNX 4)

Synopsis:

```c
#include <unistd.h>

char* qnx_crypt( const char* key,
                  const char* salt );
```

Arguments:

- **key**: A NUL-terminated string (normally a password typed by a user).
- **salt**: A two-character string chosen from the set `[a-zA-Z0-9./]`. This function doesn’t validate the values for `salt`, and values outside this range may cause undefined behavior. This string is used to perturb the algorithm in one of 4096 different ways.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `qnx_crypt()` function performs password encryption. It’s a variant of the standard `crypt()` function that uses an encryption similar to, but not compatible with, the Data Encryption Standard (DES) encryption. This function is provided for compatibility with QNX 4.

The `qnx_crypt()` function checks only the first eight characters of `key`. 
qnx_crypt()

Returns:
A pointer to the encrypted value, or NULL on failure.

Examples:
```
#include <unistd.h>

int main(int argc, char **argv) {
    char salt[3];
    char string[20];
    char *result;

    strcpy(string, "thomasf");
    salt[0] = 'a';
    salt[1] = 'B';
    salt[2] = 0;

    result = qnx_crypt(string, salt);
    printf("Result is [%s] --> [%s] \n", string, result);
    return 0;
}
```

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:
The return value points to static data that's overwritten by each call to qnx_crypt().
See also:

crypt(), encrypt(), getpass(), setkey()

login in the Utilities Reference
qsort()
Sort an array

Synopsis:

```c
#include <stdlib.h>

void qsort( void* base,
            size_t num,
            size_t width,
            int (*compare)( const void*, const void* ) );
```

Arguments:

- `base`: A pointer to the array that you want to sort.
- `num`: The number of elements in the array.
- `width`: The size of each element, in bytes.
- `compare`: A pointer to a function that compares two entries. It’s called with two arguments that point to elements in the array. The comparison function must return an integer less than, equal to, or greater than zero if the first argument is less than, equal to, or greater than the second argument.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `qsort()` function sorts the `base` array using the comparison function specified by `compare`. The array must have at least `num` elements, each of `width` bytes.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* some_strs[] = { "last", "middle", "first" };

int compare( const void* op1, const void* op2 )
{
    const char **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;

    return( strcmp( *p1, *p2 ) );
}

int main( void )
{
    qsort( some_strs,
           sizeof( some_strs ) / sizeof( char * ),
           sizeof(char *),
           compare );

    printf( "%s %s %s\n",
            some_strs[0], some_strs[1], some_strs[2] );

    return EXIT_SUCCESS;
}
```

produces the output:

first last middle

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
See also:

`bsearch()`
Raccept()

Accept a connection on a socket (via a SOCKS server)

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

int Raccept( int s,
             struct sockaddr * addr,
             int * addrlen );
```

Arguments:

- `s`: A socket that’s been created with `socket()`.
- `addr`: A result parameter that’s filled in with the address of the connecting entity, as known to the communications layer. The exact format of the `addr` parameter is determined by the domain in which the connection was made.
- `addrlen`: A value-result parameter. It should initially contain the amount of space pointed to by `addr`; on return it contains the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

Library:

- `libsocks`

Use the `-l socks` option to `qcc` to link against this library.

Description:

The `Raccept()` function is a cover function for `accept()` — the difference is that `Raccept()` does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.
Returns:

A descriptor for the accepted socket, or -1 if an error occurs (errno is set).

Classification:

SOCKS

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

accept(), Rbind(), Rconnect(), Rgetsockname(), Rlisten(), Rcmd(), Rselect(), SOCKSinit()  

SOCKS — A Basic Firewall
**Synopsis:**

```c
#include <signal.h>

int raise( int condition );
```

**Arguments:**

*condition*  
The signal that you want to raise. For more information, see `signal()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `raise()` function generates the signal specified by `condition`. Calling `raise()` is equivalent to calling:

```c
pthread_kill(pthread_self(), condition);
```

Use `SignalAction()` or `signal()` to specify the actions to take when a signal is received.

**Returns:**

0 if the specified `condition` is sent, or nonzero if an error occurs (`errno` is set).

The `raise()` function doesn’t return if the action for that signal is to terminate the program or to transfer control using the `longjmp()` function.

**Errors:**

- `EAGAIN` Insufficient system resources are available to deliver the signal.
- `EINVAL` The value of `condition` isn’t a valid signal number.
Examples:

Wait until a SIGINT signal is received. The signal is automatically raised on iteration 10000, or when you press Ctrl-C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

sig_atomic_t signal_count;
.sig.atomic_t signal_number;

void alarm_handler( int signum )
{
  ++signal_count;
  signal_number = signum;
}

int main( void )
{
  unsigned long i;

  signal_count = 0;
  signal_number = 0;
  signal( SIGINT, alarm_handler );

  printf("Iteration: ");
  for( i = 0; i < 100000; ++i ) {
      printf( "\b\b\b\b\b\b\b%*d", 5, i );
      if( i == 10000 ) raise( SIGINT );
      }
  if( signal_count > 0 ) break;

  if( i == 100000 ) {
      printf( "\nNo signal was raised.\n" );
  } else if( i == 10000 ) {
      printf( "\nSignal %d was raised by the "
            "raise() function.\n", signal_number );
  } else {
      printf( "\nUser raised signal %d.\n", 
             signal_number );
      }

  return EXIT_SUCCESS;
}
```
raise()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation</td>
<td>No</td>
</tr>
<tr>
<td>handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

signal(), SignalAction()
Generate a pseudo-random integer

Synopsis:

```c
#include <stdlib.h>

int rand( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `rand()` function computes a pseudo-random integer in the range 0 to RAND_MAX. You can start the sequence at different values by calling `srand()`.

The `rand_r()` function is a thread-safe version of `rand()`.

Returns:

A pseudo-random integer.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int i;
    for( i=1; i < 10; ++i ){
        printf( "%d\n", rand() );
    }
    return EXIT_SUCCESS;
}
```
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

\( lrand48() \), \( nrand48() \), \( rand\_r() \), \( srand() \)
**Synopsis:**
```
#include <stdlib.h>

int rand_r( unsigned int* seed );
```

**Arguments:**
- `seed` A pointer to the seed for the sequence of pseudo-random numbers. If you call `rand_r()` with the same initial value for the `seed`, the same sequence is generated.

**Library:**
- `libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `rand_r()` function computes, in a thread-safe manner, a sequence of pseudo-random integers in the range 0 through RAND_MAX.

**Returns:**
A pseudo-random integer.

**Classification:**
POSIX 1003.1 TSF

**Safety**
- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

`rand()`, `srand()`
random()

Generate a pseudo-random number from the default state

Synopsis:

```c
#include <stdlib.h>

long random( void );
```

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `random()` function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to 231-1. The period of this random-number generator is approximately $16 \times (2^{31}-1)$. The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

Use this function in conjunction with the following:

- `initstate()` Initialize the state of the pseudo-random number generator.
- `setstate()` Specify the state of the pseudo-random number generator.
- `srandom()` Set the seed used by the pseudo-random number generator.

The `random()` and `srandom()` functions have (almost) the same calling sequence and initialization properties as `rand()` and `srand()` The
difference is that `rand()` produces a much less random sequence. In fact, the low dozen bits generated by `rand()` go through a cyclic pattern. All the bits generated by `random()` are usable. For example,

```
random()&01
```

produces a random binary value.

Unlike `srand()`, `srandom()` doesn’t return the old seed because the amount of state information used is much more than a single word. The `initstate()` and `setstate()` routines are provided to deal with restarting/changing random number generators. With 256 bytes of state information, the period of the random-number generator is greater than 269.

Like `rand()`, `random()` produces by default a sequence of numbers that can be duplicated by calling `srandom()` with 1 as the seed.

If `initstate()` hasn’t been called, `random()` behaves as though `initstate()` had been called with `seed=1` and `size=128`.

If `initstate()` is called with `size` less than 8, `random()` uses a simple linear congruential random number generator.

**Returns:**

The generated pseudo-random number.

**Examples:**

See `initstate()`.

**Classification:**

POSIX 1003.1 XSI

**Safety**

- Cancellation point: No
- Interrupt handler: No
random()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

drand48(), initstate(), rand(), setstate(), srand(), srandom()
**Rbind()**

*Bind a name to a socket (via a SOCKS server)*

**Synopsis:**

```c
#include <sys/types.h>
#include <sys/socket.h>

int Rbind( int s,
           const struct sockaddr * name,
           int namelen );
```

**Arguments:**

- **s**
  - The file descriptor to be bound.

- **name**
  - A pointer to the `sockaddr` structure that holds the address to be bound to the socket. The socket length and format depend upon its address family.

- **namelen**
  - The length of the `sockaddr` structure pointed to by `name`.

**Library:**

```c
libsocks
```

Use the `-l socks` option to `qcc` to link against this library.

**Description:**

The `Rbind()` function is a cover function for `bind()` — the difference is that `Rbind()` does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.

**Returns:**

- **0** Success.
- **-1** An error occurred (`errno` is set).
**Rbind()**

**Classification:**

SOCKS

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`bind() Raccept(), Rconnect(), Rgetsockname(), Rlisten(), Rcmd(), Rselect(), SOCKSinit()`

SOCKS — A Basic Firewall
Synopsis:

```c
#include <unistd.h>

int rcmd( char ** ahost,
          unsigned short inport,
          const char * locuser,
          const char * remuser,
          const char * cmd,
          int * fd2p);
```

Arguments:

- **ahost**: The name of the host that you want to execute the command on. If the function can find the host, it sets *ahost to the standard name of the host.
- **inport**: The well-known Internet port on the host, where the server resides.
- **locuser**: The user ID on the local machine.
- **remuser**: The user ID on the remote machine.
- **cmd**: The command that you want to execute.
- **fd2p**: See below.

Library:

- **libsocket**

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `rcmd()` function is used by the superuser to execute a command, `cmd`, on a remote machine using an authentication scheme based on reserved port numbers. The `rshd` server (among others) uses the `rcmd()`, `rresvport()`, and `ruserok()` functions.

The `rcmd()` function looks up the host *ahost by means of `gethostbyname()`, and returns `-1` if the host doesn’t exist. Otherwise,
*ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*. If the connection succeeds, a SOCK_STREAM socket in the Internet domain is returned to the caller and given to the remote command as standard input and standard output.

<table>
<thead>
<tr>
<th>If <em>fd2p</em> is:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonzero</td>
<td>An auxiliary channel to a control process is set up, and a descriptor for it is placed in <em>fd2p</em>. The control process will return diagnostic output from the command (unit 2) on this channel and will accept bytes as signal numbers to be forwarded to the command’s process group.</td>
</tr>
<tr>
<td>Zero</td>
<td>The standard error (unit 2 of the remote command) is made the same as the standard output and no provision is made for sending arbitrary signals to the remote process (although you may be able to get its attention by using out-of-band data).</td>
</tr>
</tbody>
</table>

The protocol is described in detail in *rshd* in the *Utilities Reference*.

**Returns:**

A valid socket descriptor; or -1 if an error occurs and a message is printed to standard error.

**Errors:**

The error code EAGAIN is overloaded to mean “All network ports in use.”

**Classification:**

Unix
rcmd()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`gethostbyname()`, `rresvport()`, `ruserok()`

`rlogin`, `rlogind`, `rsh`, `rshd` in the *Utilities Reference*
Rconnect()  © 2007, QNX Software Systems GmbH & Co. KG.
Initiate a connection on a socket (via a SOCKS server)

Synopsis:
```c
#include <sys/types.h>
#include <sys/socket.h>

int Rconnect( int s,
              const struct sockaddr * name,
              int namelen);
```

Arguments:
- `s` The descriptor of the socket on which to initiate the connection.
- `name` The name of the socket to connect to for a SOCK_STREAM connection.
- `namelen` The length of the `name`, in bytes.

Library:
```
libsocks
```
Use the `-l socks` option to qcc to link against this library.

Description:
The `Rconnect()` function is a cover function for `connect()` — the difference is that `Rconnect()` does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.

Returns:
- 0 Success.
- -1 An error occurred (`errno` is set).
Classification:

SOCKS

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`connect()`, `Raccept()`, `Rbind()`, `Rgetsockname()`, `Rlisten()`, `Rcmd()`, `Rselect()`, `SOCKSinit()`

SOCKS — A Basic Firewall
rdchk()
Check to see if a read is likely to succeed

Synopsis:

```c
#include <unix.h>

int rdchk( int fd );
```

Arguments:

- `fd` The file descriptor that you want to check.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

> This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `rdchk()` function checks to see if a read from the file descriptor, `fd`, is likely to succeed.

Returns:

The number of characters waiting to be read, or -1 if an error occurred (`errno` is set).

Errors:

- `ENOTTY` The `fd` argument isn’t the file descriptor for a character device.

Classification:

- Unix
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`tcitschars()`
**re_comp()**  
Compile a regular expression

**Synopsis:**
```
#include <unix.h>

char *re_comp( char *s);
```

**Arguments:**
- `s`  
  A string that contains the regular expression that you want to compile. This string must end with a null byte and may include newline characters. If this argument is NULL, the current regular expression remains unchanged.

**Library:**
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**
The `re_comp()` function converts a regular expression string (RE) into an internal form suitable for pattern matching. Use this function with `re_exec()`.

The `re_comp()` and `re_exec()` functions support simple regular expressions. The regular expressions of the form `{m}`, `{m,}`, or `{m,n}` aren’t supported.

For better portability, use `regcomp()`, `regerror()`, and `regexec()` instead of these functions.
Returns:

NULL if the string pointed to by s was successfully converted. Otherwise, a pointer to one of the following error message strings is returned:
- No previous regular expression
- Regular expression too long
- unmatched \(
- missing ]
- too many \() pairs
- unmatched \)

Classification:

Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

re_exec(), regcomp(), regerror(), regexec()
grep in the Utilities Reference
Execute a regular expression

Synopsis:

```c
#include <unix.h>

int re_exec( char *s );
```

Arguments:

- `s` A pointer to the string that you want to compare to the current regular expression. This string must end with a null byte and may include newline characters.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `re_exec()` function compares the string pointed to by the string argument with the last regular expression passed to `re_comp()`.

The `re_comp()` and `re_exec()` functions support simple regular expressions. The regular expressions of the form `\{m\}`, `\{m,\}`, or `\{m,n\}` aren’t supported.

For better portability, use `regcomp()`, `regerror()`, and `regexec()` instead of these functions.

Returns:

- `1` The string matches the last compiled regular expression.
- `0` The string doesn’t match the last compiled regular expression.
-1 The compiled regular expression is invalid (indicating an internal error).

**Classification:**

Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`re_comp()`, `regcomp()`, `regerror()`, `regexec()`

`grep` in the *Utilities Reference*
**Synopsis:**

```c
#include <unistd.h>

ssize_t read(int fildes, void* buf, size_t nbyte);
```

**Arguments:**

- `fildes` The descriptor of the file that you want to read from.
- `buf` A pointer to a buffer where the function can store the data that it reads.
- `nbyte` The number of bytes that you want to read.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `read()` function attempts to read `nbyte` bytes from the file associated with the open file descriptor, `fildes`, into the buffer pointed to by `buf`.

If `nbyte` is zero, `read()` returns zero, and has no other effect.

On a regular file or other file capable of seeking, `read()` starts at a position in the file given by the file offset associated with `fildes`. Before successfully returning from `read()`, the file offset is incremented by the number of bytes actually read.

The `read()` call ignores advisory locks that you may have set with `fcntl()`.
On a file not capable of seeking, `read()` starts at the current position. When `read()` returns successfully, its return value is the number of bytes actually read and placed in the buffer. This number will never be greater than `nbyte`, although it may be less than `nbyte` for one of the following reasons:

- The number of bytes left in the file is less than `nbyte`.
- The `read()` request was interrupted by a signal.
- The file is a pipe (or FIFO) or a special file, and has fewer than `nbyte` bytes immediately available for reading. For example, reading from a file associated with a terminal may return one typed line of data.

If `read()` is interrupted by a signal before it reads any data, it returns a value of -1 and sets `errno` to EINTR. However, if `read()` is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

No data is transferred past the current end-of-file. If the starting position is at or after the end-of-file, `read()` returns zero. If the file is a device special file, the result of subsequent calls to `read()` will work, based on the then current state of the device (that is, the end of file is transitory).

If the value of `nbyte` is greater than INT_MAX, `read()` returns -1 and sets `errno` to EINVAL. See `<limits.h>`.

When attempting to read from an empty pipe or FIFO:

1. If no process has the pipe open for writing, `read()` returns 0 to indicate end-of-file.
2. If a process has the pipe open for writing, and O_NONBLOCK is set, `read()` returns -1, and `errno` is set to EAGAIN.
3. If a process has the pipe open for writing, and O_NONBLOCK is clear, `read()` blocks until some data is written, or the pipe is closed by all processes that had opened it for writing.
When attempting to read from a file (other than a pipe or FIFO) that support nonblocking reads and has no data currently available:

1. If O_NONBLOCK is set, read() returns -1, and errno is set to EAGAIN.

2. If O_NONBLOCK is clear, read() blocks until some data is available.

3. The O_NONBLOCK flag has no effect if some data is available.

If you call read() on a portion of a file, prior to the end-of-file, that hasn’t been written, it returns bytes with the value zero.

If read() succeeds, the st_atime field of the file is marked for update.

Returns:

The number of bytes actually read, or -1 (errno is set).

Errors:

EAGAIN The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the read operation.

EBADF The file descriptor, fildes, isn’t a valid file descriptor open for reading.

EINTR The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.

EIO A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSYS The read() function isn’t implemented for the filesystem specified by filedes.
EOVERFLOW The file is a regular file and an attempt is made to read at or beyond the offset maximum associated with the file.

Examples:

```c
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
    int fd;
    int size_read;
    char buffer[80];

    /* Open a file for input */
    fd = open( "myfile.dat", O_RDONLY );

    /* Read the text */
    size_read = read( fd, buffer, sizeof( buffer ) );

    /* Test for error */
    if( size_read == -1 ) {
        perror( "Error reading myfile.dat" );
        return EXIT_FAILURE;
    }

    /* Close the file */
    close( fd );

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1 XSI

Safety

Cancellation point Yes

continued…
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`close()`, `creat()`, `dup()`, `dup2()`, `errno`, `fcntl()`, `lseek()`, `open()`, `pipe()`,
`readblock()`, `readv()`, `select()`, `write()`, `writev()`
read_main_config_file()

Read the snmpd.conf file

Synopsis:

```
#include <snmp/snmp_api.h>

int read_main_config_file(
    struct snmpd_conf_data * info);
```

Arguments:

`info` A pointer to a `snmpd_conf_data` structure that the function can fill with the configuration information. For more information about this structure, see below.

Library:

`libsnmp`

Use the `-l snmp` option to `qcc` to link against this library.

Description:

The `read_main_config_file()` function fills the `info` structure with data from the `snmpd.conf` file (see the Utilities Reference). This information is useful if you wish to know what configuration information the SNMP agent was started with.

The string pointers in this structure, if not NULL, point to strings obtained by using `malloc()`; you can free them by calling `free()`.

If the data for a member of the structure isn’t available, the structure member isn’t modified. You should use `memset()` to set the structure to 0 before calling `read_main_config_file()`.

To locate the `snmpd.conf` file, this function first checks the `SNMPCONFIGFILE` environment variable. If this isn’t found, the default, `snmpd.conf`, is used. If the specified file couldn’t be accessed, the structure members aren’t updated.

The `snmpd_conf_data` structure is defined in `<snmp_api.h>`, and contains the following members:
struct snmpd_conf_data{
    char* main_config_fname;
    char* party_conf_fname;
    char* view_conf_fname;
    char* context_conf_fname;
    char* acl_conf_fname;
    char* sysContact;
    char* sysLocation;
    char* sysName;
    char* private_community;
    char* public_community;
    char* trap_sink;
    char* trap_community;
    int conf_authentraps;
};

The members of this structure are:

main_config_fname
    snmpd.conf file location.

party_conf_fname
    party.conf file location.

view_conf_fname
    view.conf file location.

context_conf_fname
    context.conf file location.

acl_conf_fname
    acl.conf file location.

sysContact
    system.sysContact string.

sysLocation
    system.sysLocation string.

sysName
    system.sysName string.

private_community
    Private level community string name to use.

public_community
    Public level community string name to use.
read_main_config_file()

*trap_sink*  Destination host to send trap messages.

*trap_community*  Community name to use for trap messages.

*conf_authentrap*  Enable authentication traps (1 means enable, 2 means disable).

**Returns:**

0  Success.

-1  An error occurred (*errno* is set).

**Errors:**

ENOENT  The *snmpd.conf* file wasn’t found.

**Files:**

*snmpd.conf*  Default SNMP configuration file. For more information, see the *Utilities Reference*.

**Environment variables:**

SNMPCONFIGFILE  
Location of the SNMP configuration file.

**Classification:**

SNMP

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

continued…
read_main_config_file() © 2007, QNX Software Systems GmbH & Co. KG.

Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

snmp_close(), snmp_free_pdu(), snmp_open(), snmp_pdu,
snmp_pdu_create(), snmp_read(), snmp_select_info(), snmp_send(),
snmp_session, snmp_timeout()

snmpd, snmpd.conf in the Utilities Reference
Synopsis:

```c
#include <unistd.h>

int readblock( int fd,
               size_t blksize,
               unsigned block,
               int numblks,
               void *buff );
```

Arguments:

- `fd`: The file descriptor for the file you want to read from.
- `blksize`: The number of bytes in each block of data.
- `block`: The block number from which to start reading.
- `numblks`: The number of blocks to read.
- `buff`: A pointer to a buffer where the function can store the data that it reads.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `readblock()` function reads `numblks` blocks of data from the file associated with the open file descriptor `fd`, into the buffer pointed to by `buff`, starting at block number `block` (blocks are numbered starting at 0). The `blksize` argument specifies the size of a block, in bytes.

This function is useful for direct access to raw blocks on a block special device (for example, raw disk blocks) but may also be used for high-speed access to database files, for example. (The speed gain is through the combined seek/read implicit in this call, and the ability to
transfer more than the `read()` function’s limit of INT_MAX bytes at a time.)

If `numblks` is zero, `readblock()` returns zero and has no other results.

On successful completion, `readblock()` returns the number of blocks actually read and placed in the buffer. This number is never greater than `numblks`. The value returned may be less than `numblks` if one of the following occurs:

- The number of blocks left before the end-of-file is less than `numblks`.
- The process requests more blocks than implementation limits allow to be read in a single atomic operation.
- A read error occurred after reading at least one block.

If a read error occurs on the first block, `readblock()` returns -1 and sets `errno` to EIO.

**Returns:**

The number of blocks actually read. If an error occurs, it returns -1, sets `errno` to indicate the error, and the contents of the buffer pointer to by `buff` are left unchanged.

**Errors:**

- **EBADF**  The `fd` argument isn’t a valid file descriptor open for reading a block-oriented device.
- **EIO**    A physical read error occurred on the first block.
- **EINVAL** The starting position is invalid (0 or negative) or beyond the end of the disk.

**Classification:**

QNX Neutrino
readblock()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

writeblock()
**readcond()**

Read data from a terminal device

**Synopsis:**

```c
#include <unistd.h>

int readcond( int fd, 
              void * buf, 
              int n, 
              int min, 
              int time, 
              int timeout );
```

**Arguments:**

- `fd` The file descriptor associated with the terminal device that you want to read from.
- `buf` A pointer to a buffer into which `readcond()` can put the data.
- `n` The maximum number of bytes to read.
- `min`, `time`, `timeout`
  
  When used in RAW mode, these arguments override the behavior of the `MIN` and `TIME` members of the terminal’s `termios` structure. For more information, see below.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `readcond()` function reads up to `n` bytes of data from the terminal device indicated by `fd` into the buffer pointed to by `buf`.

This function is an alternative to the `read()` function for terminal devices, providing additional arguments for timed read operations. These additional arguments can be used to minimize overhead when dealing with terminal devices.
The three arguments \((\text{min}, \text{time}, \text{and} \text{timeout})\), when used on terminal devices in RAW mode, override the behavior of the \textit{MIN} and \textit{TIME} elements of the currently defined \textit{termios} structure (for the duration of this call only). The \textit{termios} structure also defines a forwarding character (in \texttt{c_cc[VFWD]}) that can be used to bypass \textit{min}, \textit{time} and \textit{timeout}.

The normal case of a simple read by an application would block until at least one character was available.

**Conditions that satisfy an input request.**

In the case where multiple conditions are specified, the read will be satisfied when any one of them is satisfied.

**\textit{MIN}**

The qualifier \textit{MIN} is useful when an application has knowledge of the number of characters it expects to receive.

Any protocol that knows the character count for a frame of data can use \textit{MIN} to wait for the entire frame to arrive. This significantly reduces IPC and process scheduling. \textit{MIN} is often used in conjunction with \textit{TIME} or \textit{TIMEOUT}. \textit{MIN} is part of the POSIX standard.

**\textit{TIME}**

The qualifier \textit{TIME} is useful when an application is receiving streaming data and wishes to be notified when the data stops or pauses. The pause time is specified in 1/10 of a second. \textit{TIME} is part of the POSIX standard.
**TIMEOUT**

The qualifier `TIMEOUT` is useful when an application has knowledge of how long it should wait for data before timing out. The timeout is specified in 1/10 of a second.

Any protocol that knows the character count for a frame of data it expects to receive can use `TIMEOUT`. This in combination with the baud rate allows a reasonable guess to be made when data should be available. It acts as a deadman timer to detect dropped characters. It can also be used in interactive programs with user input to timeout a read if no response is available within a given time.

`TIMEOUT` is a QNX extension and isn’t part of the POSIX standard.

**FORWARD**

The qualifier `FORWARD` is useful when a protocol is delimited by a special framing character. For example, the PPP protocol used for TCP/IP over a serial link start and end its packets with a framing character. When used in conjunction with `TIMEOUT`, the `FORWARD` character can greatly improve the efficiency of a protocol implementation. The protocol process will receive complete frames, rather than character by character. In the case of a dropped framing character, `TIMEOUT` or `TIME` can be used to quickly recover.

This greatly minimizes the amount of IPC work for the OS and results in a much lower processor utilization for a given TCP/IP data rate. It is interesting to note that PPP doesn’t contain a character count for its frames. Without the data-forwarding character, an implementation would be forced to read the data one character at a time.

`FORWARD` is a QNX extension and isn’t part of the POSIX standard.

To enable the `FORWARD` character, you must set the VFWD character in the `c_cc` member of the `termios` structure:

```c
/* PPP forwarding character */
const char fwd_char = 0x7e;

#include <termios.h>
```

: 
```c
int fd;
struct termios termio;

:
:

tcgetattr(fd, &termio);
termio.c_cc[VFWD] = fwd_char;
tcsetattr(fd, TCSANOW, &termio);
```

The following table summarizes the interaction of `min`, `time`, and `timeout`:

<table>
<thead>
<tr>
<th>min</th>
<th>time</th>
<th>timeout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Returns immediately with as many bytes as are currently available (up to ( n ) bytes).</td>
</tr>
<tr>
<td>( M )</td>
<td>0</td>
<td>0</td>
<td>Return with up to ( n ) bytes only when at least ( M ) bytes are available.</td>
</tr>
<tr>
<td>0</td>
<td>( T )</td>
<td>0</td>
<td>Return with up to ( n ) bytes when at least one byte is available, or ( T ) * .1 sec has expired.</td>
</tr>
<tr>
<td>( M )</td>
<td>( T )</td>
<td>0</td>
<td>Return with up to ( n ) bytes when either ( M ) bytes are available, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds ( T ) * .1 sec.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>( t )</td>
<td>RESERVED.</td>
</tr>
<tr>
<td>( M )</td>
<td>0</td>
<td>( t )</td>
<td>Return with up to ( n ) bytes when ( t ) * .1 sec has expired, or ( M ) bytes are available.</td>
</tr>
<tr>
<td>0</td>
<td>( T )</td>
<td>( t )</td>
<td>RESERVED.</td>
</tr>
</tbody>
</table>

continued...
readcond()

<table>
<thead>
<tr>
<th>min</th>
<th>time</th>
<th>timeout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>T</td>
<td>t</td>
<td>Return with up to ( n ) bytes when ( M ) bytes are available, or ( t \times 0.1 ) sec has expired and no characters are received, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds ( T \times 0.1 ) sec.</td>
</tr>
</tbody>
</table>

Note that when \( timeout \) is zero, the behavior of \( min \) and \( time \) is exactly the same as the behavior of the \texttt{MIN} and \texttt{TIME} parameters of the \texttt{termios} controlling structure. Thus, \texttt{readcond()} can be used as a higher speed alternative to consecutive calls of \texttt{tcgetattr()}, \texttt{tcsetattr()}, and \texttt{read()} when dealing with RAW terminal I/O.

The \((M, 0, t)\) case is useful for communications protocols that cannot afford to block forever waiting for data that may never arrive.

The \((M, T, t)\) case is provided to permit \texttt{readcond()} to return when a burst of data ends (as in the \((M, T, 0)\) case), but also to return if no burst at all is detected within a reasonable amount of time.

**Returns:**

The number of bytes read, or -1 if an error occurs (\texttt{errno} is set).

**Errors:**

- **EAGAIN** The O\_NONBLOCK flag is set on this \texttt{fd}, and the process would have been blocked in trying to perform this operation.
- **EBADF** The argument \texttt{fd} is invalid or file isn’t opened for reading.
- **EINVAL** The \texttt{readcond()} call was interrupted by the process being signalled.
- **EIO** This process isn’t currently able to read data from this \texttt{fd}.
ENOSYS  This function isn’t supported for this fd.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

erro, read(), tcgetattr(), tcsetattr(), termios
**readdir()**

*Read a directory entry*

**Synopsis:**

```c
#include <dirent.h>

struct dirent * readdir( DIR * dirp );
```

**Arguments:**

- `dirp` A pointer to the directory stream to be read.

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `readdir()` function reads the next directory entry from the directory specified by `dirp`, which is the value returned by a call to `opendir()`.

You can call `readdir()` repeatedly to list all of the entries contained in the directory specified by the pathname given to `opendir()`. The `closedir()` function must be called to close the directory stream and free the memory allocated by `opendir()`.

The `<dirent.h>` file defines the `struct dirent` and the `DIR` type used by the `readdir()` family of functions.

The result of using a directory stream after one of the `exec*()` or `spawn*()` family of functions is undefined. After a call to `fork()`, either the parent or the child (but not both) can continue processing the directory stream, using the `readdir()` and `rewinddir()` functions. If both the parent and child processes use these functions, the result is undefined. Either (or both) processes may use `closedir()`.

The `<dirent.h>` file also defines the following macros for accessing extra data associated with the `dirent` structure:
readdir()

_DEXTRA_FIRST(pdirent)

Get a pointer to the first block of data associated with the structure pointed to by pdirent.

_DEXTRA_NEXT(last)

Get the block of data that follows the block pointed to by last.

_DEXTRA_VALID(extra, pdirent)

Evaluates to 1 if extra is a pointer to a valid block of data associated with the structure pointed to by pdirent.

You can use these macros to traverse the data associated with the dirent structure like this:

```c
for( extra = _DEXTRA_FIRST(dirent);
      _DEXTRA_VALID(extra, dirent);
     extra = _DEXTRA_NEXT(extra)) {
    switch(extra->d_type) {
      /* No data */
      case _DTYPE_NONE :
        break;
      /* Data includes information as returned by stat() */
      case _DTYPE_STAT :
        break;
      /* Data includes information as returned by lstat() */
      case _DTYPE_LSTAT :
        break;
      ...
    }
  }
```

Returns:

A pointer to a struct dirent object for success, or NULL if the end of the directory stream is encountered or an error occurs (errno is set).

Errors:

EBADF The dirp argument doesn’t refer to an open directory stream.

EOVERFLOW One of the values in the structure to be returned can’t be represented correctly.
Examples:

Get a list of files contained in the directory `/home/fred`:

```c
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

int main( void )
{
    DIR* dirp;
    struct dirent* direntp;

    dirp = opendir( "./home/fred" );
    if( dirp != NULL ) {
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;
            printf( "%s\n", direntp->d_name );
        }
    }
    closedir( dirp );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

closedir(), errno, lstat(), opendir(), readdir_r(), rewinddir(),
seekdir(), telldir(), stat()
**Synopsis:**

```c
#include <sys/types.h>
#include <dirent.h>

int readdir_r( DIR * dirp,
               struct dirent * entry,
               struct direct ** result );
```

**Arguments:**

- `dirp` A pointer to the directory stream to be read.
- `entry` A pointer to a `dirent` structure where the function can store the directory entry.
- `result` The address of a location where the function can store a pointer to the information found.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

If `_POSIX_THREAD_SAFE_FUNCTIONS` is defined, `readdir_r()` initializes the `dirent` structure referenced by `entry` with the directory entry at the current position in the directory stream referred to by `dirp`, and stores a pointer to this structure in `result`.

The storage pointed by `entry` must be large enough for a `dirent` structure with the `s_name` member an array of char containing at least `NAME_MAX` plus one element.
The `struct dirent` structure doesn’t include space for the pathname. You must provide it:

```c
struct dirent *entry;
entry = malloc( sizeof(*entry) + NAME_MAX + 1 );
```

**Returns:**

EOK Success.

On failure, `errno` is set.

**Errors:**

- EBADF The `dirp` argument doesn’t refer to an open directory stream.
- EOVERFLOW One of the values in the structure to be returned can’t be represented correctly.

**Classification:**

POSIX 1003.1 TSF

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

`closedir()`, `errno`, `opendir()`, `readdir()`, `seekdir()`, `telldir()`, `rewinddir()`
Place the contents of a symbolic link into a buffer

Synopsis:

```c
#include <unistd.h>

int readlink( const char* path,  
                char* buf,        
                size_t bufsiz );
```

Arguments:

- `path` The name of the symbolic link.
- `buf` A pointer to a buffer where the function can store the contents of the symbolic link (i.e. the path linked to).
- `bufsiz` The size of the buffer.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `readlink()` function places the contents of the symbolic link named by `path` into the buffer pointed to by `buf`, which has a size of `bufsiz`. The contents of the returned symbolic link doesn’t include a NULL terminator. Its length must be determined from the `stat` structure returned by `lstat()`, or by the return value of the `readlink()` call.

If `readlink()` is successful, up to `bufsiz` bytes from the contents of the symbolic link are placed in `buf`.

Returns:

The number of bytes placed in the buffer, or -1 if an error occurs (`errno` is set).
Errors:

**EACCES** Search permission is denied for a component of the *path* prefix.

**EINVAL** The named file isn’t a symbolic link.

**ELOOP** A loop exists in the symbolic links encountered during resolution of the path argument, and more than SYMLOOP_MAX symbolic links were encountered.

**ENAMETOOLONG** A component of the *path* exceeded NAME_MAX characters, or the entire pathname exceeded PATH_MAX characters.

**ENOENT** The named file doesn’t exist.

**ENOSYS** Links aren’t supported by the resource manager associated with path.

**ENOTDIR** A component of the *path* prefix named by path isn’t a directory.

Examples:

```c
/*
 * Read the contents of the named symbolic links
 */
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char buf[PATH_MAX + 1];

int main( int argc, char** argv )
{
    int n;
    int len;
    int ecode = 0;

    for( n = 1; n < argc; ++n ) {
        if(( len = readlink( argv[n], buf, PATH_MAX )) == -1) {
            perror( argv[n] );
        }
    }
    return ecode;
}
```
readlink()

```c
ecode++; } 
else {
   buf[len] = '\0';
   printf( "%s -> %s\n", argv[n], buf );
} 

return( ecode );
```

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `lstat()`, `symlink()`
readv()

Synopsis:

```c
#include <sys/uio.h>

ssize_t readv(int fildes, const iov_t* iov, int iovcnt);
```

Arguments:

- `fildes` The descriptor of the file that you want to read from.
- `iov` An array of `iov_t` objects where the function can store the data that it reads.
- `iovcnt` The number of entries in the `iov` array. The maximum number of entries is `UIO_MAXIOV`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `readv()` function attempts to read from the file associated with the open file descriptor, `fildes`, placing the data into `iovcnt` buffers specified by the members of the `iov` array: `iov[0], iov[1], ..., iov[iovcnt-1].`

On a regular file or other file capable of seeking, `readv()` starts at a position in the file given by the file offset associated with `fildes`. Before successfully returning from `readv()`, the file offset is incremented by the number of bytes actually read.

The `iov_t` structure contains the following members:

- `iov_base` The base address of a memory area where data should be placed.
The `readv()` function always fills one buffer completely before proceeding to the next.

The `readv()` call ignores advisory locks that may have been set by the `fcntl()` function.

On a file not capable of seeking, `readv()` starts at the current position.

When `readv()` returns successfully, its return value is the number of bytes actually read and placed in the buffer. This number will never be greater than the combined sizes of the `iov` buffers, although it may be less for one of the following reasons:

- The number of bytes left in the file is less than the combined size of the `iov` buffers.
- The `readv()` request was interrupted by a signal.
- The file is a pipe (or FIFO) or a special file, and has fewer bytes immediately available for reading. For example, reading from a file associated with a terminal may return one typed line of data.

If `readv()` is interrupted by a signal before it reads any data, it returns a value of -1 and sets `errno` to EINTR. However, if `readv()` is interrupted by a signal after it has successfully read some data, it returns the number of bytes read.

No data is transferred past the current end-of-file. If the starting position is at or after the end-of-file, `readv()` returns zero. If the file is a device special file, the result of subsequent calls to `readv()` will work, based on the then current state of the device (that is, the end of file is transitory).

When attempting to read from an empty pipe or FIFO:

1. If no process has the pipe open for writing, `readv()` returns 0 to indicate end-of-file.
If a process has the pipe open for writing, and O_NONBLOCK is set, `readv()` returns -1 and sets `errno` to EAGAIN.

If a process has the pipe open for writing, and O_NONBLOCK is clear, `read()` blocks until some data is written, or the pipe is closed by all processes that had opened it for writing.

When attempting to read from a file (other than a pipe or FIFO) that supports nonblocking reads and has no data currently available:

1. If O_NONBLOCK is set, `readv()` returns -1 and sets `errno` to EAGAIN.
2. If O_NONBLOCK is clear, `readv()` blocks until some data is available.
3. The O_NONBLOCK flag has no effect if some data is available.

If you call `readv()` on a portion of a file, prior to the end-of-file, that hasn’t been written, it returns bytes with the value zero.

If `readv()` succeeds, the `st_atime` field of the file is marked for update.

**Returns:**

The number of bytes read, or -1 if an error occurred (`errno` is set).

**Errors:**

- **EAGAIN** The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the read operation.
- **EBADF** The file descriptor, `fildes`, isn’t a valid file descriptor open for reading.
- **EINTR** The read operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.
readv()

EINVAL  The `iovcnt` argument is less than or equal to 0, or greater than UIO_MAXIOV.
EIO    A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.
ENOSYS The `readv()` function isn’t implemented for the filesystem specified by `filedes`.
EOVERFLOW  The file is a regular file and an attempt is made to read at or beyond the offset maximum associated with the file.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

close(), creat(), dup(), dup2(), errno, fcntl(), lseek(), open(), pipe(), read(), readblock(), select(), write(), writev()
realloc()

Allocate, reallocate, or free a block of memory

Synopsis:

```c
#include <stdlib.h>

void* realloc( void* old_blk, size_t size );
```

Arguments:

- `old_blk`: A pointer to the block of memory to be allocated, reallocated, or freed.
- `size`: The new size, in bytes, for the block of memory.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `realloc()` function allocates, reallocates, or frees the block of memory specified by `old_blk` based on the following rules:

- If `old_blk` is NULL, a new block of memory of `size` bytes is allocated.
- If the `size` is zero, the `free()` function is called to release the memory pointed to by `old_blk`.
- Otherwise, `realloc()` reallocates space for an object of `size` bytes by:
  - shrinking the size of the allocated memory block `old_blk` when `size` is smaller than the current size of `old_blk`
  - extending the allocated size of the allocated memory block `old_blk` if there is a large enough block of unallocated memory immediately following `old_blk`
realloc()

- allocating a new block with the appropriate size, and copying the contents of old_blk to this new block

The realloc() function allocates memory from the heap.

Because it’s possible that a new block will be allocated, any pointers into the old memory could be invalidated. These pointers will point to freed memory, with possible disastrous results, when a new block is allocated.

The realloc() function returns NULL when the memory pointed to by old_blk can’t be reallocated. In this case, the memory pointed to by old_blk isn’t freed, so be careful to maintain a pointer to the old memory block so it can be freed later.

In the following example, buffer is set to NULL if the function fails, and won’t point to the old memory block. If buffer is your only pointer to the memory block, then you have lost access to this memory.

```c
buffer = (char*) realloc( buffer, 100 );
```

Returns:

A pointer to the start of the allocated memory, or NULL if an error occurred (errno is set).

Errors:

- ENOMEM Not enough memory.
- EOK No error.

Examples:

```c
#include <stdlib.h>
#include <malloc.h>

int main( void )
{
    char* buffer;
```
char* new_buffer;

buffer = (char*) malloc( 80 );
new_buffer = (char*) realloc( buffer, 100 );
if( new_buffer == NULL ) {
    /* not able to allocate larger buffer */
    return EXIT_FAILURE;
} else {
    buffer = new_buffer;
}

return EXIT_SUCCESS;

Classification:
ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
calloc(), free(), malloc(), sbrk()
realpath()
Resolve a pathname

Synopsis:
#include <stdlib.h>

char * realpath( const char * pathname, 
char * resolved_name );

Arguments:

pathname       The path name that you want to resolve.

resolved_name  A pointer to a buffer where the function can store the 
resolved name.

Library:

libc

Use the -l c option to qcc to link against this library. This library is 
usually included automatically.

Description:

The realpath() function resolves all symbolic links, extra slash (/) 
characters and references to ./ and ../ in pathname, and copies 
the resulting absolute pathname into the memory referenced by 
resolved_name.

To determine the size of the buffer pointed to by resolved_name, call 
_fpathconf() or pathconf() with an argument of _PC_PATH_MAX.

This function resolves both absolute and relative paths and returns the 
absolute pathname corresponding to pathname. All but the last 
component of pathname must exist when realpath() is called.

Returns:

A pointer to resolved_name, or NULL if an error occurs, in which 
case resolved_name contains the pathname that caused the problem.
The `realpath()` function may fail and set the external variable `errno` for any of the errors specified for the library functions `chdir()`, `close()`, `lstat()`, `open()`, `readlink()` and `getcwd()`.

**Classification:**

POSIX 1003.1 XSI

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

This implementation of `realpath()` differs slightly from the Solaris implementation. QNX always returns absolute pathnames, whereas the Solaris implementation, under certain circumstances, returns a relative `resolved_name` when given a relative pathname.

**See also:**

`getcwd()`
recv()

Receive a message from a socket

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv( int s,
              void * buf,
              size_t len,
              int flags );
```

Arguments:

- `s` The descriptor for the socket; see `socket()`.
- `buf` A pointer to a buffer where the function can store the message.
- `len` The size of the buffer.
- `flags` A combination formed by ORing one or more of the values:
  - `MSG_OOB` — process out-of-band data. This flag requests receipt of out-of-band data that wouldn’t be received in the normal data stream. You can’t use this flag with protocols that place expedited data at the head of the normal data queue.
  - `MSG_PEEK` — peek at the incoming message. This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
  - `MSG_WAITALL` — wait for full request or error. This flag requests that the operation block until the full request is satisfied. But the call may still return less data than requested if a signal is caught, if an error or disconnect occurs, or if the next data to be received is of a different type than that returned.
recv()

The MSG_WAITALL flag isn’t supported by the tiny TCP/IP stack. For more information, see npm-ttcpip.so in the Utilities Reference.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The recv() function receives a message from a socket. It’s normally used only on a connected socket — see connect() — and is identical to recvfrom() with a zero from parameter.

This routine returns the length of the message on successful completion. If a message is too long for the supplied buffer, buf, then excess bytes might be discarded, depending on the type of socket that the message is received from; see socket().

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking—see ioctl()—in which case -1 is returned and the external variable errno is set to EWOULDBLOCK. Normally, the receive calls return any data available, up to the requested amount, rather than wait for the full amount requested; this behavior is affected by the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in getsockopt().

You can use select() to determine when more data is to arrive.

Returns:

The number of bytes received, or -1 if an error occurs (errno is set).

Errors:

EBADF       Invalid descriptor s.
EFAULT      The receive buffer is outside the process’s address space.
recv()

EINTR The receive was interrupted by delivery of a signal before any data was available.

ENOTCONN The socket is associated with a connection-oriented protocol and hasn’t been connected; see connect() and accept().

EWOULDBLOCK Either the socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See also:

connect(), ioctl(), getsockopt(), read(), recvfrom(), recvmsg(), select(), socket()
recvfrom()

Receive a message from the socket at a specified address

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom( int s,
                  void * buff,
                  size_t len,
                  int flags,
                  struct sockaddr * from,
                  socklen_t * fromlen );
```

Arguments:

- `s` The descriptor for the socket; see `socket()`.
- `buf` A pointer to a buffer where the function can store the message.
- `len` The size of the buffer.
- `flags` A combination formed by ORing one or more of the values:
  - `MSG_OOB` — process out-of-band data. This flag requests receipt of out-of-band data that wouldn’t be received in the normal data stream. You can’t use this flag with protocols that place expedited data at the head of the normal data queue.
  - `MSG_PEEK` — peek at the incoming message. This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
  - `MSG_WAITALL` — wait for full request or error. This flag requests that the operation block until the full request is satisfied. But the call may still return less data than requested if a signal is caught, if an error or disconnect occurs, or if the next data to be received is of a different type than that returned.

© 2007, QNX Software Systems GmbH & Co. KG.

C Library — P to R September 10, 2007
recvfrom()

The MSG_WAITALL flag isn’t supported by the tiny TCP/IP stack. For more information, see `npm-ttcpip.so` in the Utilities Reference.

- **from**: NULL, or a pointer to a `sockaddr` object where the function can store the source address of the message.

- **fromlen**: A pointer to a `socklen_t` object that specifies the size of the `from` buffer. The function stores the actual size of the address in this object.

**Library:**

libsocket

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `recvfrom()` routine receives a message from the socket, `s`, whether or not it’s connection-oriented.

If `from` is nonzero, and the socket is connectionless, the source address of the message is filled in. The parameter `fromlen` is a value-result parameter, initialized to the size of the buffer associated with `from`, and modified on return to indicate the actual size of the stored address.

This routine returns the length of the message on successful completion. If a message is too long for the supplied buffer, `buf`, excess bytes may be discarded depending on the type of socket that the message is received from—see `socket()`.

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking—see `ioctl()`—in which case `recvfrom()` returns -1 is returned and sets the external variable `errno` to EWOULDBLOCK. Normally, the receive calls return any data available, up to the requested amount, rather than wait for the full amount requested; this behavior is affected by the socket-level options `SO_RCVLOWAT` and `SO_RCVTIMEO` described in `getsockopt()`.
You can use `select()` to determine when more data is to arrive.

**Returns:**

The number of bytes received, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EBADF** Invalid descriptor `s`.
- **EFAULT** The receive buffer pointer(s) point outside the process’s address space.
- **EINTR** The receive was interrupted by delivery of a signal before any data was available.
- **ENOTCONN** The socket is associated with a connection-oriented protocol and hasn’t been connected; see `connect()` and `accept()`.
- **EWOULDBLOCK** Either the socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.

**Classification:**

- POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
See also:

recv(), recvmsg(), select()
recvmsg()

Receive a message and its header from a socket

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvmsg( int s,
    struct msghdr * msg,
    int flags );
```

Arguments:

- `s` The descriptor for the socket; see `socket()`.
- `msg` A pointer to a `msghdr` structure where the function can store the message header; see below.
- `len` The size of the buffer.
- `flags` A combination formed by ORing one or more of the values:
  - `MSG_OOB` — process out-of-band data. This flag requests receipt of out-of-band data that wouldn’t be received in the normal data stream. You can’t use this flag with protocols that place expedited data at the head of the normal data queue.
  - `MSG_PEEK` — peek at the incoming message. This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.
  - `MSG_WAITALL` — wait for full request or error. This flag requests that the operation block until the full request is satisfied. But the call may still return less data than requested if a signal is caught, if an error or disconnect occurs, or if the next data to be received is of a different type than that returned.
recvmsg()

The MSG_WAITALL flag isn’t supported by the tiny TCP/IP stack. For more information, see npm-ttcpi.so in the Utilities Reference.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The recvmsg() routine receives a message from a socket, s, whether or not it’s connection-oriented.

The recvmsg() call uses a msghdr structure to minimize the number of directly supplied parameters. This structure, defined in <sys/socket.h>, has the following form:

```c
struct msghdr {
    caddr_t msg_name; /* optional address */
    u_int msg_namelen; /* size of address */
    struct iovec *msg_iov; /* scatter/gather array */
    u_int msg_iovlen; /* # elements in msg_iov */
    caddr_t msg_control; /* ancillary data, see below */
    u_int msg_controllen; /* ancillary data buffer len */
    int msg_flags; /* flags on received message */
};
```

The msg_name and msg_namelen parameters specify the address (source address for recvmsg(); destination address for sendmsg()) if the socket is unconnected; the msg_name parameter may be given as a null pointer if no names are desired or required.

The msg_iov and msg_iovlen parameters describe scatter-gather locations, as discussed in read().

The msg_control parameter, whose length is determined by msg_controllen, points to a buffer for other protocol-control related messages or for other miscellaneous ancillary data. The messages are of the form:

```c
struct cmsghdr {
    u_int cmsg_len; /* data byte count, including hdr */
    /* ancillary data */
};
```
Currently, the tiny TCP/IP stack doesn't support ancillary data. The `msg_controllen` member of `struct msghdr` must be 0.

The `msg_flags` field is set on return according to the message received:

- **MSG_CTRUNC**: Indicates that some control data was discarded due to lack of space in the buffer for ancillary data.
- **MSG_EOR**: Indicates end-of-record; the data returned completed a record.
- **MSG_OOB**: Indicates that expedited or out-of-band data was received.
- **MSG_TRUNC**: Indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied.

**Returns:**

The number of bytes received, or -1 if an error occurs (`errno` is set).

**Errors:**

- **ENOMEM**: Not enough memory.

**Classification:**

POSIX 1003.1
\textbf{Safety}

\begin{tabular}{ll}
Cancellation point & Yes \\
Interrupt handler  & No \\
Signal handler    & No \\
Thread            & Yes \\
\end{tabular}

See also:

\textit{recv()}, \textit{recvfrom()}, \textit{sendmsg()}
### Synopsis:

```c
#include <regex.h>

int regcomp( regex_t * preg,
              const char * pattern,
              int cflags );
```

### Arguments:

- **preg**: A pointer to a regex_t object where the function can store the compiled regular expression.
- **pattern**: The regular expression that you want to compile; see below.
- **cflags**: A bitwise inclusive OR of zero or more of the following flags:
  - REG_EXTENDED — use Extended Regular Expressions.
  - REG_ICASE — ignore differences in case.
  - REG_NEWLINE — treat <newline> as a regular character.
  - REG_NOSUB — report only success/failure in regexec().

### Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

---

This function is in `libc.a`, but not in `libc.so` (in order to save space).
Description:

The `regcomp()` function prepares the regular expression, `preg`, for use by the function `regexec()`, from the specification `pattern` and `cflags`. The member `re_nsub` of `preg` is set to the number of subexpressions in `pattern`.

The functions that deal with regular expressions (`regcomp()`, `regerror()`, `regexec()`, and `regfree()`) support two classes of regular expressions, the Basic and Extended Regular Expressions. These classes are rigorously defined in IEEE P1003.2, Regular Expression Notation.

Basic Regular Expressions

The Basic Regular Expressions are composed of these terms:

- $x$ x at end of line ($ must be the last term).
- ^x x at beginning of line (^ must be first the term).
- x* Zero or more occurrences of x.
- . Any single character (except newline).
- c The character c.
- xc x followed by the character c.
- cx Character c followed by x.
- [cd] The characters c or d.
- [c-d] All characters between c and d, inclusive.
- [^c] Any character but c.
- [[:classname:]] Any of the following classes:
  - alnum
  - alpha
**Extended Regular Expressions**

The Extended Regular Expressions also include:

- **x+** One or more occurrences of \( x \).
- **x?** Zero or one occurrences of \( x \).
- **(x)** Subexpression \( x \) (for precedence handling).
- **x/y** Expression \( x \) OR \( y \).
regcomp()

Returns:

0       Success.

<>0     An error occurred (use regerror() to get an explanation).

Examples:

/*
   The following example prints out all lines
   from FILE "f" that match "pattern".
*/
#include <stdio.h>
#include <regex.h>
#include <limits.h>

#define BUFFER_SIZE 512

void grep( char* pattern, FILE* f )
{
    int t;
    regex_t re;
    char buffer[BUFFER_SIZE];

    if ((t=regcomp( &re, pattern, REG_NOSUB )) != 0) {
        regerror(t, &re, buffer, sizeof buffer);
        fprintf(stderr,"grep: %s (%s)\n",buffer,pattern);
        return;
    }
    while( fgets( buffer, BUFFER_SIZE, f ) != NULL ) {
        if( regexec( &re, buffer, 0, NULL, 0 ) == 0 ) {
            fputs( buffer, stdout );
        }
    }
    regfree( &re );
}

Classification:

POSIX 1003.1

Safety

Cancellation point   No

continued…
**regcomp()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Contributing author:**

Henry Spencer. For copyright information, see Third-Party Copyright Notices in this reference.

**See also:**

regerror(), regexec(), regfree()
Synopsis:

```
#include <regex.h>

size_t regerror( int err,
               const regex_t * reg,
               char * buf,
               size_t len );
```

Arguments:

- `err` The value returned by a previous call to `regcomp()` or `regexec()`.
- `reg` A pointer to the `regex_t` object for the regular expression that you provided to the failed call to `regcomp()` or `regexec()`.
- `buf` A pointer to a buffer where the function can store the explanation.
- `len` The length of the buffer, in characters.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `regerror()` function provides a string explaining an error code returned by `regcomp()` or `regexec()`. The string is copied into `buf` for up to `len` characters.
Returns:

The number of characters copied into the buffer.

Examples:

See `regcomp()`.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Contributing author:

Henry Spencer. For copyright information, see Third-Party Copyright Notices in this reference.

See also:

`regcomp()`, `regexec()`, `regfree()`
Synopsis:

```c
#include <regex.h>

int regexec( const regex_t * preg,
            const char * string,
            size_t nmatch,
            regmatch_t * pmatch,
            int eflags );
```

Arguments:

- **preg** A pointer to the `regex_t` object for the regular expression that you want to execute. You must have compiled the expression by calling `regcomp()`.
- **string** The string that you want to match against the regular expression.
- **nmatch** The maximum number of matches to record in `pmatch`.
- **pmatch** An array of `regmatch_t` objects where the function can record the matches; see below.
- **eflags** Execution parameters to `regexec()`. For example, you may need to call `regexec()` multiple times if the line you’re processing is too large to fit into `string`. The `eflags` argument is the bitwise inclusive OR of zero or more of the following flags:
  - REG_NOTBOL — the `string` argument doesn’t point to the beginning of a line.
  - REG_NOTEOL — the end of `string` isn’t the end of a line.

Library:

```
l libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
This function is in \texttt{libc.a}, but not in \texttt{libc.so} (in order to save space).

\section*{Description:}

The \texttt{regexec()} function compares string against the compiled regular expression \texttt{preg}. If \texttt{regexec()} finds a match it returns zero; otherwise, it returns nonzero.

The \texttt{preg} argument represents a compiled form of either a Basic Regular Expression or Extended Regular Expression. These classes are rigorously defined in IEEE P1003.2, Regular Expression Notation, and are summarized in the documentation for \texttt{regcomp()}.

The \texttt{regexec()} function records the matches in the \texttt{pmatch} array, with \texttt{nmatch} specifying the maximum number of matches to record. The \texttt{regmatch\_t} structure is defined as:

\begin{verbatim}
typedef struct {
  regoff_t rm_so;
  regoff_t rm_eo;
} regmatch_t;
\end{verbatim}

The members are:

\begin{itemize}
  \item \texttt{rm\_sp} \quad The byte offset from the beginning of the string to the beginning of the matched substring.
  \item \texttt{rm\_ep} \quad One greater than the offset from the beginning of the string to the end of the matched substring.
\end{itemize}

The offsets in \texttt{pmatch[0]} identify the substring corresponding to the entire expression, while those in \texttt{pmatch[1...nmatch]} identify up to the first \texttt{nmatch} subexpressions. Unused elements of the \texttt{pmatch} array are set to -1.
You can disable the recording of substrings by either specifying REG_NOSUB in `regcomp()`, or by setting `nmatch` to zero.

**Returns:**

0       The *string* argument matches *preg*.

<>0     A match wasn’t found, or an error occurred (use `regerror()` to get an explanation).

**Examples:**

See `regcomp()`.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Contributing author:**

Henry Spencer. For copyright information, see Third-Party Copyright Notices in this reference.
See also:

`regcomp()`, `regerror()`, `regfree()`
Synopsis:

```c
#include <regex.h>

void regfree( regex_t * preg );
```

Arguments:

`preg`  A pointer to the `regex_t` object for the regular expression that you want to free; see `regcomp()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `regfree()` function releases all memory allocated by `regcomp()` associated with `preg`.

Examples:

See `regcomp()`.

Classification:

POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No

continued...
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Contributing author:**

Henry Spencer. For copyright information, see Third-Party Copyright Notices in this reference.

**See also:**

`regcomp()`, `regerror()`, `regexec()`
remainder(), remainderf()

Compute the floating point remainder

Synopsis:

```c
#include <math.h>

double remainder( double x,  
                 double y );

float remainderf( float x,  
                   float y );
```

Arguments:

- `x` The numerator of the division.
- `y` The denominator.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `remainder()` and `remainderf()` functions return the floating point remainder \( r = x - ny \), where \( y \) is nonzero. The value \( n \) is the integral value nearest the exact value \( x/y \). When \( |n - x/y| = \frac{1}{2} \), the value \( n \) is chosen to be even.

The behavior of `remainder()` is independent of the rounding mode.

Returns:

The floating point remainder \( r = x - ny \), where \( y \) is nonzero.
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set \texttt{errno} to 0, call the function, and then check \texttt{errno} again. These functions don’t change \texttt{errno} if no errors occurred.

\textbf{Classification:}

ANSI, POSIX 1003.1

\begin{center}
\begin{tabular}{l c}
\textbf{Safety} & \\
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\end{tabular}
\end{center}

\textbf{See also:}

\texttt{drem()}, \texttt{modf()}
remove()
Remove a link to a file

Synopsis:
#include <stdio.h>

int remove( const char * filename );

Arguments:
filename The path to the file that you want to delete.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The remove() function removes a link to a file:

- If the filename names a symbolic link, remove() removes the link, but doesn’t affect the file or directory that the link goes to.

- If the filename isn’t a symbolic link, remove() removes the link and decrements the link count of the file that the link refers to.

  If the link count of the file becomes zero, and no process has the file open, then the space that the file occupies is freed, and no one can access the file anymore.

  If one or more processes have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed.

The remove() function also works on directories; however, rmdir() is more efficient.
remove()

Returns:

0 The operation was successful.
Nonzero The operation failed (errno is set).

Errors:

EACCES Search permission is denied for a component of filename, or write permission is denied on the directory containing the link to be removed.
EBUSY The directory named by filename can’t be unlinked because it’s being used by the system or another process, and the target filesystem or resource manager considers this to be an error.
ENAMETOOLONG The filename argument exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.
ENOENT The named file doesn’t exist, or filename is an empty string.
ENOSYS The remove() function isn’t implemented for the filesystem specified by filename.
ENOTDIR A component of filename isn’t a directory.
EPERM The file named by filename is a directory, and either the calling process doesn’t have the appropriate privileges, or the target filesystem or resource manager prohibits using remove() on directories.
EROFS The directory entry to be unlinked resides on a read-only filesystem.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( remove( "vm.tmp" ) ) {
        puts( "Error removing vm.tmp!" );
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno`, `rmdir()`, `unlink()`
rename()

Synopsis:

```c
#include <stdio.h>

int rename( const char* old,
            const char* new );
```

Arguments:

- `old` The path to the file that you want to rename. If the path doesn’t include a directory, `rename()` looks for the file in the current working directory.

- `new` The new name for the file. If the path doesn’t include a directory, `rename()` creates the file in the current working directory.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `rename()` function changes the name of the file indicated by `old` to the name given in `new`.

If `new` identifies an existing file or empty directory, `rename()` overwrites it.

Returns:

- 0 Success.
- Nonzero An error occurred (`errno` is set).

C Library — P to R

September 10, 2007
rename()

Errors:

EACCESS The calling program doesn’t have permission to search one of the components of either path prefix, or one of the directories containing old or new denies write permission.

EBUSY The directory named by old or new can’t be renamed because another process is using it.

EEXIST The file specified by new is a directory that contains files.

EINVAL The new directory pathname contains the old directory.

EISDIR The file specified by new is a directory and old is a file.

ELOOP Too many levels of symbolic links.

EMLINK The file named by old is a directory, and the link count of the parent directory of new would exceed LINK_MAX.

ENAMETOOLONG The length of old or new exceeds PATH_MAX.

ENOENT The old file doesn’t exist, or old or new is an empty string.

ENOSPC The directory that would contain new can’t be extended.

ENOSYS The rename() function isn’t implemented for the filesystem specified in old or new.

ENOTDIR A component of either path prefix isn’t a directory, or old is a directory and new isn’t.

ENOTEMPTY The file specified by new is a directory that contains files.
**rename()**

The `rename()` would affect files on a read-only filesystem.

EXDEV The files or directories named by old and new are on different filesystems.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( rename( "old.dat", "new.dat" ) ) {
        puts( "Error renaming old.dat to new.dat." );
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`
res_init()  
Initialize the Internet domain name resolver routines

Synopsis:
#include <sys/types.h>  
#include <netinet/in.h>  
#include <arpa/nameser.h>  
#include <resolv.h>

int res_init( void );

Library:  
libsocket
Use the -l socket option to qcc to link against this library.

Description:  
The resolver routines are used for making, sending, and interpreting  
query and reply messages with Internet domain name servers.

The res_init() routine reads the resolver configuration file (if one is  
present; see /etc/resolv.conf in the Utilities Reference) to get  
the default domain name, search list, and Internet address of the local  
name servers. If no server is configured, the host running the resolver  
is tried. If not specified in the configuration file, the current domain  
name is defined by the hostname; the domain name can be overridden  
by the environment variable LOCALDOMAIN. Initialization  
normally occurs on the first call to one of the resolver routines.

Resolver configuration

Global configuration and state information used by these routines is  
kept in the __res_state structure _res, which is defined in  
<resolv.h>. Since most of the values have reasonable defaults, you  
can generally ignore them.

The _res.options member is a simple bit mask that contains the  
bitwise OR of the enabled options. The following options are defined  
in <resolv.h>:

RES_DEBUG      Print debugging messages.
RES_DEFNAMES
If this option is set, res_search() appends the default domain name to single-component names (those that don’t contain a dot). This option is enabled by default.

RES_DNSRCH
If this option is set, res_search() searches for hostnames in the current domain and in parent domains. This is used by the standard host lookup routine, gethostbyname(). This option is enabled by default.

RES_INIT
True if the initial name server address and default domain name are initialized (i.e. res_init() has been called).

RES_RECURSE
Set the recursion-desired bit in queries. This is the default. Note that res_send() doesn’t do iterative queries — it expects the name server to handle recursion.

RES_STAYOPEN
Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the mode you normally use.

RES_USEVC
Instead of UDP datagrams, use TCP connections for queries.

Returns:

0 Success.
Nonzero An error occurred.
Errors:

See herror().

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, LOCALDOMAIN contains a domain name that overrides the current domain name.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

dn_comp(), dn_expand(), gethostbyname(), res_mkquery(), res_query(), res_querydomain(), res_search(), res_send()

hostname, /etc/resolv.conf in the Utilities Reference

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035
res_mkquery() © 2007, QNX Software Systems GmbH & Co. KG.

Construct an Internet domain name query

Synopsis:

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_mkquery( int op,
                 const char * dname,
                 int class,
                 int type,
                 const u_char * data,
                 int datalen,
                 const u_char * newrr,
                 u_char * buf,
                 int buflen );
```

Arguments:

- **op** Usually QUERY, but it can be also IQUERY or NS_NOTIFY_OP. Note that not all of the query types defined in `<arpa/nameser.h>` are supported.

- **dname** The domain name for the query.

- **class** The class of information that you want; one of:
  - C_IN — ARPA Internet.
  - C_CHAOS — Chaos net (MIT).
  - C_HS — Hesiod name server (MIT).
  - C_ANY — any class.
You typically use C_IN.

- **type** The type of information that you want. You typically use T_PTR, but you can use any of the T_* constants defined in `<arpa/nameser.h>`.

- **data** NULL, or a pointer to resource record data.

- **datalen** The length of the data.
**res_mkquery()**

`newrr` Currently unused. This argument is intended for making update messages.

`buf` A pointer to a buffer where the function can build the query.

` buflen` The length of the buffer.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `res_mkquery()` function is a low-level routine that’s used by `res_query()` to construct an Internet domain name query. This routine constructs a standard query message and places it in `buf`. It returns the size of the query, or -1 if the query is larger than `buflen`.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers. Global configuration and state information used by the resolver routines is kept in the structure `_res`. For more information on the options, see `res_init()`.

**Returns:**

The size of the prepared query, in bytes, or -1 if an error occurs.

**Files:**

`/etc/resolv.conf`

Resolver configuration file.

**Environment variables:**

**LOCALDOMAIN**

When set, `LOCALDOMAIN` contains a domain name that overrides the current domain name.
**res_mkquery()**

**Classification:**
Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

dn_comp(), dn-expand(), gethostbyname(), res_init(), res_query(),
res_querydomain(), res_search(), res_send()

hostname, /etc/resolv.conf in the Utilities Reference
RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035
res_query()

Query the local Internet domain name server

Synopsis:
```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query( const char * dname,
    int class,
    int type,
    u_char * answer,
    int anslen );
```

Arguments:
- **dname**: The fully qualified domain name that you want to query.
- **class**: The class of information that you want; one of:
  - C_IN — ARPA Internet.
  - C_CHAOS — Chaos net (MIT).
  - C_HS — Hesiod name server (MIT).
  - C_ANY — any class.
  You typically use C_IN.
- **type**: The type of information that you want. You typically use T_PTR, but you can use any of the T_* constants defined in <arpa/nameser.h>.
- **answer**: A pointer to a buffer where the function can store the answer to the query.
- **anslen**: The length of the buffer.

Library:
```
libsocket
```
Use the -l socket option to qcc to link against this library.
Description:

The `res_query()` function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, waits for a response, and makes preliminary checks on the reply. The query requests information of the specified `type` and `class` for the specified fully qualified domain name `dname`. The reply message is left in the `answer` buffer with length `anslen` supplied by the caller.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers. Global configuration and state information used by the resolver routines is kept in the structure `_res`. For more information on the options, see `res_init()`.

The `res_query()` function uses the following lower-level routines:

- `res_mkquery()` constructs a standard query message.
- `res_send()` sends the preformatted query and returns an answer.
- `dn_comp()` compresses a domain name.
- `dn_expand()` expands the compressed domain name to a full domain name.

Returns:

The length of a reply message, in bytes, or -1 if an error occurs (`h_errno` is set).

Errors:

See `herror()`.

Files:

`/etc/resolv.conf`

Resolver configuration file.
Environment variables:

LOCALDOMAIN

When set, LOCALDOMAIN contains a domain name that overrides the current domain name.

Classification:

Unix

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

dn_comp(), dn_expand(), gethostbyname(), herror(), res_init(), res_mkquery(), res_querydomain(), res_search(), res_send()

hostname, /etc/resolv.conf in the Utilities Reference

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035
**res_querydomain()**

Query the local Internet domain name server

**Synopsis:**
```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_querydomain( const char * name,
                     const char * domain,
                     int class,
                     int type,
                     u_char * answer,
                     int anslen );
```

**Arguments:**

- **name** The host name that you want to query.
- **domain** The domain name that you want to query.
- **class** The class of information that you want; one of:
  - C_IN — ARPA Internet.
  - C_CHAOS — Chaos net (MIT).
  - C_HS — Hesiod name server (MIT).
  - C_ANY — any class.
  You typically use C_IN.
- **type** The type of information that you want. You typically use
  T_PTR, but you can use any of the T_* constants defined in
  <arpa/nameser.h>.
- **answer** A pointer to a buffer where the function can store the
  answer to the query.
- **anslen** The length of the buffer.
res_querydomain()

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The res_querydomain() function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, waits for a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the host specified by concatenating name and domain. The trailing dot is removed from name if domain is 0.

The reply message is left in the answer buffer with length anslen supplied by the caller.

Returns:

0 Success.
-1 An error occurred.

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, LOCALDOMAIN contains a domain name that overrides the current domain name.

Classification:

Unix
res_querydomain()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

res_init(), res_query()
res_search()

Query a local server, using search options

Synopsis:

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_search( const char * dname,
                int class,
                int type,
                u_char * answer,
                int anslen );
```

Arguments:

dname     The fully qualified domain name that you want to query.
class     The class of information that you want; one of:
              • C_IN — ARPA Internet.
              • C_CHAOS — Chaos net (MIT).
              • C_HS — Hesiod name server (MIT).
              • C_ANY — any class.

You typically use C_IN.
type     The type of information that you want. You typically use T_PTR, but you can use any of the T_ constants defined in `arpa/nameser.h`.
answer    A pointer to a buffer where the function can store the answer to the query.
anslen    The length of the buffer.

Library:

libsocket

Use the `-l socket` option to `gcc` to link against this library.
Description:

The res_search() routine makes an Internet domain name search. Like res_query(), res_search() makes a query and waits for a response. But it also implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information used by the resolver routines is kept in the structure _res. For more information on the options, see res_init().

Returns:

The length of a reply message, in bytes, or -1 if an error occurs (h_errno is set).

Errors:

See herror.

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, LOCALDOMAIN contains a domain name that overrides the current domain name.

Classification:

Unix
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

- `dn_comp()`, `dn_expand()`, `gethostbyname()`, `herror()`, `res_init()`, `res_mkquery()`, `res_query()`, `res_querydomain()`, `res_send()`
- `hostname`, `/etc/resolv.conf` in the *Utilities Reference*
- *RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035*
res_send()

Send a preformatted Internet domain name query

Synopsis:

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send( const u_char * msg, 
              int msglen, 
              u_char * answer, 
              int anslen );
```

Arguments:

- `msg` The preformatted Internet domain name query that you want to send.
- `msglen` The length of the message.
- `answer` A pointer to a buffer where the function can store the answer to the query.
- `anslen` The length of the buffer.

Library:

```
libsocket
```

Use the -l socket option to gcc to link against this library.

Description:

The `res_send()` function is a low-level routine that’s used by `res_query()` to send a preformatted Internet domain name query and return an answer. It calls `res_init()` if RES_INIT isn’t set, sends the query to the local name server, and handles timeouts and retries.

The resolver routines are used for making, sending, and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information used by the resolver routines is kept in the structure `res`. For more information on the options, see `res_init()`.
Returns:

The length of a reply message, in bytes; or -1 if an error occurs.

Errors:

ECONNREFUSED

No name servers found.

ETIMEDOUT

No answer obtained.

Files:

/etc/resolv.conf

Resolver configuration file.

Environment variables:

LOCALDOMAIN

When set, LOCALDOMAIN contains a domain name that overrides the current domain name.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

dn_comp(), dn_expand(), gethostbyname(), res_init(),
res_mkquery(), res_query(), res_querydomain(), res_search()
hostname, /etc/resolv.conf in the Utilities Reference

RFC 974, RFC 1032, RFC 1033, RFC 1034, RFC 1035
resmgr_attach()

Attach a path to the pathname space

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int resmgr_attach (  
    dispatch_t *dpp,  
    resmgr_attr_t *attr,  
    const char *path,  
    enum _file_type file_type,  
    unsigned flags,  
    const resmgr_connect_funcs_t *connect_funcs,  
    const resmgr_io_funcs_t *io_funcs,  
    RESMGR_HANDLE_T *handle );
```

Arguments:

- **dpp**
  A dispatch handle created by `dispatch_create()`.

- **attr**
  A pointer to a `resmgr_attr_t` structure that defines attributes for the resource manager; see below.

- **path**
  NULL, or the path that you want to attach the resource manager to; see below.

- **file_type**
  The file type; one of the following (defined in `<sys/ftype.h>`):
  - `_FTYPE_ANY` — the path name can be anything.
  - `_FTYPE_LINK` — reserved for the Process Manager.
  - `_FTYPE_MOUNT` — receive mount requests on the path (`path` must be NULL).
  - `_FTYPE_MQUEUE` — reserved for a message-queue manager.
  - `_FTYPE_PIPE` — reserved for a pipe manager.
  - `_FTYPE_SEM` — reserved for a semaphore manager.
### resmgr_attach()

- **_FTYPE_SHMEM** — reserved for a shared memory object.
- **_FTYPE_SOCKET** — reserved for a socket manager.
- **_FTYPE_SYMLINK** — reserved for the Process Manager.

#### flags

Flags that control the pathname resolution:

- **_RESMGR_FLAG_AFTER**
- **_RESMGR_FLAG_BEFORE** (the default)
- **_RESMGR_FLAG_OPAQUE**
- **_RESMGR_FLAG_DIR**
- **_RESMGR_FLAG_FTYPEONLY**
- **_RESMGR_FLAG_MASK**
- **_RESMGR_FLAG_SELF**

For more information, see “Flags,” below.

#### connect_funcs

A pointer to the `resmgr_connect_funcs_t` structure that defines the POSIX-level connect functions.

#### io_funcs

A pointer to the `resmgr_io_funcs_t` structure that defines the POSIX-level I/O functions.

#### handle

A pointer to an arbitrary structure that you want to associate with the pathname you’re attaching. For most resource managers, this is an `iofunc_attr_t` structure.

### Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `resmgr_attach()` function puts the path into the general pathname space and binds requests on this path to the dispatch handle `dpp`. Most of the above file types are used for special services that have their own open function associated with them. For example, the mqueue manager specifies `file_type` as `FTYPE_MQUEUE` and `mq_open()` requests a pathname match of the same type.

Specify `_FTYPE_ANY` for normal filesystems and simple devices, such as serial ports, that don’t have their own special open type. Also if you can handle the type of service or a redirection node to a manager that does. Most resource managers are of this type.

Your resource manager won’t receive messages from an open of an inappropriate type. The following table shows the different open function types and the types of pathnames they’ll match.

<table>
<thead>
<tr>
<th>Function</th>
<th>file_type:</th>
<th>Matches pathname of type:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mq_open()</code></td>
<td><code>FTYPE_MQUEUE</code></td>
<td><code>FTYPE_ANY</code> or <code>FTYPE_MQUEUE</code></td>
</tr>
<tr>
<td><code>open()</code></td>
<td><code>FTYPE_ANY</code></td>
<td>All types</td>
</tr>
<tr>
<td><code>pipe()</code></td>
<td><code>FTYPE_PIPE</code></td>
<td><code>FTYPE_ANY</code> or <code>FTYPE_PIPE</code></td>
</tr>
<tr>
<td><code>sem_open()</code></td>
<td><code>FTYPE_SEM</code></td>
<td><code>FTYPE_ANY</code> or <code>FTYPE_SEM</code></td>
</tr>
<tr>
<td><code>shm_open()</code></td>
<td><code>FTYPE_SHMEM</code></td>
<td><code>FTYPE_ANY</code> or <code>FTYPE_SHMEM</code></td>
</tr>
<tr>
<td><code>socket()</code></td>
<td><code>FTYPE_SOCKET</code></td>
<td><code>FTYPE_ANY</code> or <code>FTYPE_SOCKET</code></td>
</tr>
</tbody>
</table>

The generic `open()` can be used to open a pathname of any type.

If you want to use the POSIX functions, we’ve provided you with the POSIX layer; to fill your connect and I/O functions tables with the default handler functions supplied by the POSIX layer library, call
iofunc_func_init(). You can then override the defaults placed in the structures with your own handlers.

In the most general case, the last argument, handle is an arbitrary structure that you wish to have associated with the pathname you’re attaching. Practically, however, we recommend that it contain the POSIX layer’s well defined attributes structure, iofunc_attr_t, because this lets you use the POSIX layer default library. You can extend the data that’s contained in the attributes structure to contain any device-specific data that you may require. This is commonly done, and is described in the “Extending Data Control Structures (DCS)” section in the Writing a Resource Manager chapter of the Programmer’s Guide.

In order to use the POSIX layer default library, the attributes structure must be bound into the Open Control Block, and you must use the POSIX layer’s iofunc_ocb_t OCB. This is described in the documentation for resmgr_open_bind(), as well as in the above reference.

resmgr_attr_t structure

You can specify attributes such as the maximum message size, number of parts (number of IOVs in context), and flags in the attr structure. The resmgr_attr_t structure looks like this:

```c
typedef struct _resmgr_attr {
    unsigned flags;
    unsigned nparts_max;
    unsigned msg_max_size;
    int (*other_func)( resmgr_context_t *, void *msg );
} resmgr_attr_t;
```

The members include:

- nparts_max The number of components to allocate for the IOV array. If you specify 0, the resource manager library bumps the value to the minimum usable by the library itself.
resmgr_attach()

msg_max_size

The minimum amount of room to reserve for receiving a message that’s allocated in resmgr_context_alloc(). If the value is too low, or you specify it as 0, resmgr_attach() picks a value that’s usable.

other_func

A pointer to a function that’s called if the resource manager receives an I/O message that it didn’t successfully handle. This function is attached only if the RESMGR_FLAG_ATTACH_OTHERFUNC flag (defined in <sys/dispatch.h>) is set.

Flags

The flags argument specifies additional information to control the pathname resolution. The flags (defined in <sys/resmgr.h>) include at least the following bits:

_RESMGR_FLAG_AFTER

Force the path to be resolved after others with the same pathname at the same mountpoint.

_RESMGR_FLAG_BEFORE

Force the path to be resolved before others with the same pathname at the same mountpoint.

_RESMGR_FLAG_DIR

Treat the pathname as a directory and allow the resolving of longer pathnames. The _IO_CONNECT message contains the pathname passed to the client open() with the matching prefix stripped off. Without this flag, the pathname is treated as a simple file requiring an exact match.
resmgr_attach()

<table>
<thead>
<tr>
<th>Attached path</th>
<th>Opened path</th>
<th>_RESMGR_FLAG_DIR set</th>
<th>_RESMGR_FLAG_DIR clear</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a/b</td>
<td>/a/b</td>
<td>Match **</td>
<td>Match **</td>
</tr>
<tr>
<td>/a/b</td>
<td>/a/b/c</td>
<td>Match c</td>
<td>No match</td>
</tr>
<tr>
<td>/a/b</td>
<td>/a/b/c/d</td>
<td>Match c/d</td>
<td>No match</td>
</tr>
<tr>
<td>/a/b</td>
<td>/a/bc</td>
<td>No match</td>
<td>No match</td>
</tr>
</tbody>
</table>

You can’t attach a directory pathname that contains, as a subset, an existing file pathname. Likewise, you can’t attach a file pathname that’s a subset of an existing directory pathname.

<table>
<thead>
<tr>
<th>Existing path</th>
<th>New path</th>
<th>New path allowed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory /a/b</td>
<td>Directory /a</td>
<td>Yes</td>
</tr>
<tr>
<td>Directory /a/b</td>
<td>Directory /a/b/c</td>
<td>Yes</td>
</tr>
<tr>
<td>File /a/b</td>
<td>Directory /a</td>
<td>Yes</td>
</tr>
<tr>
<td>File /a/b</td>
<td>Directory /a/b/c</td>
<td>No; the directory is beneath a file</td>
</tr>
<tr>
<td>Directory /a/b</td>
<td>File /a</td>
<td>No; the directory is beneath a file</td>
</tr>
<tr>
<td>Directory /a/b</td>
<td>File /a/b/c</td>
<td>Yes</td>
</tr>
<tr>
<td>File /a/b</td>
<td>File /a</td>
<td>Yes</td>
</tr>
<tr>
<td>File /a/b</td>
<td>File /a/b/c</td>
<td>Yes</td>
</tr>
</tbody>
</table>

_RESMGR_FLAG_FTYPEONLY

Handle only requests for the specific filetype indicated. The pathname must be NULL.

_RESMGR_FLAG_OPAQUE

Don’t resolve paths to mountpoints on a path shorter than this (i.e. find the longest match against all pathnames attached).
resmgr_attach()

_RESMGR_FLAG_SELF

Allow requests to resolve back to this server (a deadlock is possible).

Returns:

A unique link ID associated with this attach, or -1 on failure (errno is set).

The returned ID is needed to detach the pathname at a later time using resmgr_detach(). The ID is also passed back in the resmgr_handler() function in ctp->id.

Errors:

ENOMEM There isn’t enough free memory to complete the operation.

ENOTDIR A component of the pathname wasn’t a directory entry.

Examples:

Here’s an example of a simple single-threaded resource manager:

```c
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t connect_funcs;
static resmgr_io_funcs_t io_funcs;
static iofunc_attr_t attr;

int main(int argc, char **argv)
{
    dispatch_t *dpp;
    resmgr_attr_t resmgr_attr;
    resmgr_context_t *ctp;
    int id;

    /* initialize dispatch interface */
    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "Unable to allocate \n" );
        return -1;
    }

    /* initialize connect callback function */
    connect_funcs.connect = ...
```

September 10, 2007 C Library — P to R 2539
For more examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, and `thread_pool_create()`. For more information on writing a resource manager, see the “Writing a Resource Manager” chapter in the Programmer’s Guide.
Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

If your application calls this function, it must run as root.

See also:

dispatch_create(), iofunc_attr_init(), iofunc_attr_t, iofunc_func_init(), iofunc_ocb_t, resmgr_block(), resmgr_connect_funcs_t, resmgr_context_alloc(), resmgr_context_free(), resmgr_detach(), resmgr_handler(), resmgr_io_funcs_t

“Writing a Resource Manager” chapter of the Programmer's Guide.
resmgr_block()  © 2007, QNX Software Systems GmbH & Co. KG.

Block while waiting for a message

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

resmgr_context_t * resmgr_block
    ( resmgr_context_t * ctp );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `resmgr_block()` function waits for a message (created by a call to `resmgr_context_alloc()`) for context `ctp`.

This function is a special case of `dispatch_block()` that you should use only with a simple resource manager. If you need to attach pulses or other messages, then you should use `dispatch_block()`.

Returns:

The same pointer as `ctp`, or NULL if an error occurs (`errno` is set).

Errors:

- `EFAULT` A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender’s buffers only when `MsgReceive()` is called, a fault could occur in the sender if the sender’s
buffers are invalid. If a fault occurs when accessing the sender buffers (only) they’ll receive an EFAULT and the `MsgReceive()` won’t unblock.

EINTR The call was interrupted by a signal.

ETIMEDOUT A kernel timeout (that was set with `dispatch_timeout()`) unblocked the call.

Examples:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    resmgr_context_t *ctp;

    if ((dpp = dispatch_create()) == NULL) {
        fprintf( stderr, "%s: Unable to allocate \n dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }

    .

    ctp = resmgr_context_alloc( dpp );

    while (1) {
        if ( ( ctp = resmgr_block( ctp ) ) == NULL ) {
            fprintf(stderr, "block error\n" );
            return EXIT_FAILURE;
        }
        resmgr_handler( ctp );
    }
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

Classification:

QNX Neutrino
**resmgr_block()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

Use `resmgr_block()` only in a simple resource manager and when you don’t use `message_attach()`, `pulse_attach()`, or `select_attach()`.

**See also:**

`dispatch_block()`, `resmgr_attach()`, `resmgr_context_alloc()`, `resmgr_handler()`

“Components of a Resource Manager” section of the Writing a Resource Manager chapter in the *Programmer’s Guide.*
Synopsis:

```c
#include <sys/resmgr.h>

typedef struct _resmgr_connect_funcs {
    unsigned nfuncs;
    int (*open) (resmgr_context_t *ctp, io_open_t *msg,
                 RESMGR_HANDLE_T *handle, void *extra);
    int (*unlink) (resmgr_context_t *ctp, io.unlink_t *msg,
                   RESMGR_HANDLE_T *handle, void *reserved);
    int (*rename) (resmgr_context_t *ctp, io_rename_t *msg,
                   RESMGR_HANDLE_T *handle,
                   io_rename_extra_t *extra);
    int (*mknod) (resmgr_context_t *ctp, io_mknod_t *msg,
                   RESMGR_HANDLE_T *handle, void *reserved);
    int (*readlink) (resmgr_context_t *ctp, io_readlink_t *msg,
                     RESMGR_HANDLE_T *handle, void *reserved);
    int (*link) (resmgr_context_t *ctp, io_link_t *msg,
                 RESMGR_HANDLE_T *handle,
                 io_link_extra_t *extra);
    int (*unblock) (resmgr_context_t *ctp, io_pulse_t *msg,
                   RESMGR_HANDLE_T *handle, void *reserved);
    int (*mount) (resmgr_context_t *ctp, io_mount_t *msg,
                  RESMGR_HANDLE_T *handle,
                  io_mount_extra_t *extra);
} resmgr_connect_funcs_t;
```

Description:

The `resmgr_connect_funcs_t` structure is a table of the POSIX-level connect functions that are used by a resource manager. You can initialize this table by calling `iofunc_func_init()` and then overriding the defaults with your own functions.
This structure includes `nfuncs`, which indicates how many functions are in the table (in case the structure grows in the future), along with these functions:

<table>
<thead>
<tr>
<th>Member</th>
<th>Used to</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>open</code></td>
<td>Handle <code>_IO_CONNECT</code> messages</td>
<td><code>iofunc_open_default()</code></td>
</tr>
<tr>
<td><code>unlink</code></td>
<td>Unlink the resource</td>
<td>None</td>
</tr>
<tr>
<td><code>rename</code></td>
<td>Rename the resource</td>
<td>None</td>
</tr>
<tr>
<td><code>mknod</code></td>
<td>Create a filesystem entry point</td>
<td>None</td>
</tr>
<tr>
<td><code>readlink</code></td>
<td>Read a symbolic link</td>
<td>None</td>
</tr>
<tr>
<td><code>link</code></td>
<td>Create a symbolic link</td>
<td>None</td>
</tr>
<tr>
<td><code>unblock</code></td>
<td>Unblock the resource if an operation is aborted</td>
<td>None</td>
</tr>
<tr>
<td><code>mount</code></td>
<td>Mount a filesystem</td>
<td>None</td>
</tr>
</tbody>
</table>

**Classification:**

QNX Neutrino

**See also:**

`iofunc_func_init()`, `iofunc_open_default()`, `resmgr_io_funcs_t`

Writing a Resource Manager chapter of the QNX Neutrino Programmer’s Guide
resmgr_context_alloc()

Allocate a resource-manager context

Synopsis:

#include <sys/iofunc.h>
#include <sys/dispatch.h>

resmgr_context_t * resmgr_context_alloc
    ( dispatch_t * dpp );

Arguments:

    dpp  A dispatch handle created by dispatch_create().

Library:

    libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The resmgr_context_alloc() function returns a context that’s used for blocking and receiving messages.

This function is a special case of dispatch_context_alloc(). You should use it only when writing a simple resource manager.

Returns:

A pointer to a resmgr_context_t structure, or NULL if an error occurs (errno is set).

Errors:

    EINVAL  No resource manager events were attached to dpp.
    ENOMEM  Insufficient memory to allocate the context ctp.
resmgr_context_alloc()

© 2007, QNX Software Systems GmbH & Co. KG.

Examples:
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
dispatch_t
*dpp;
resmgr_context_t
*ctp;
if ( (dpp = dispatch_create()) == NULL ) {
fprintf( stderr, "%s: Unable to allocate \
dispatch handle.\n",argv[0] );
return EXIT_FAILURE;
}
.
.
.
if ( ( ctp = resmgr_context_alloc ( dpp )) == NULL ) {
fprintf( stderr, "Context wasn’t allocated.\n" );
return EXIT_FAILURE;
}
}

For examples using the dispatch interface, see dispatch_create(),
message_attach(), resmgr_attach(), and thread_pool_create().

Classification:
QNX Neutrino
Safety

2548

Cancellation point

Yes

Interrupt handler

No

Signal handler

No

Thread

Yes

C Library — P to R

September 10, 2007


See also:

`dispatch_context_alloc()`, `dispatch_create()`, `resmgr_attach()`, `resmgr_context_free()`, `resmgr_context_t`

“Components of a Resource Manager” in the Writing a Resource Manager chapter of the *Programmer’s Guide*
Free a resource-manager context

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void resmgr_context_free( resmgr_context_t * ctp );
```

Arguments:

- `ctp` A pointer to the `resmgr_context_t` structure that you want to free.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `resmgr_context_free()` function frees a context allocated by `resmgr_context_alloc()`.

This function is a special case of `dispatch_context_free()`. You should use it only when writing a simple resource manager.

Examples:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    resmgr_context_t *ctp;

    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
```
::

if ( (ctp = resmgr_context_alloc (dpp)) == NULL ) {
    fprintf(stderr, "Context wasn’t allocated.\n" );
    return EXIT_FAILURE;
}

::

resmgr_context_free (ctp);

For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

resmgr_context_alloc(), resmgr_context_t

“Components of a Resource Manager” in the Writing a Resource Manager chapter of the Programmer’s Guide
resmgr_context_t

Context information that's passed between resource-manager functions

Synopsis:

```c
#include <sys/resmgr.h>

typedef struct _resmgr_context {
    int rcvid;
    struct _msg_info info;
    resmgr_iomsgs_t *msg;
    dispatch_t *dpp;
    int id;
    unsigned tid;
    unsigned msg_max_size;
    int status;
    int offset;
    int size;
    iov_t iov[1];
} resmgr_context_t;
```

Description:

The `resmgr_context_t` structure defines context information that's passed to resource-manager functions.

The members include:

- **rcvid**: The receive ID to use for messages to and from the client.
- **info**: A pointer to a `_msg_info` structure that contains information about the message received by the resource manager.
- **msg**: A pointer to the message received by the resource manager, expressed as a union of all the possible message types.
- **dpp**: The dispatch handle, created by `dispatch_create()`.
- **id**: The link Id, returned by `resmgr_attach()`.
- **tid**: Not used; always zero.
- **msg_max_size**: The minimum amount of space reserved for receiving a message.
resmgr_context_t

status A place to store the status of the current operation. Always use _RESMGR_STATUS() to set this member.

offset The offset, in bytes, into the client’s message. You’ll use this when working with combine messages.

size The number of valid bytes in the message area.

iov An I/O vector where you can place the data that you’re returning to the client.

Classification:

QNX Neutrino

See also:

dispatch_create(), _msg_info, MsgInfo(), resmgr_attach(), resmgr_context_alloc(), resmgr_context_free(),
_RESMGR_STATUS()
resmgr_detach()  © 2007, QNX Software Systems GmbH & Co. KG.

Remove a pathname from the pathname space

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int resmgr_detach( dispatch_t * dpp,  
                  int id,  
                  unsigned flags );
```

Arguments:

- **dpp** A dispatch handle created by `dispatch_create()`.
- **id** The link ID that `resmgr_attach()` returned.
- **flags** Flags that affect the operation. The possible flags (defined in `<sys/dispatch.h>` and `<sys/resmgr.h>`) are:
  - `_RESMGR_DETACH_ALL` — detach the name from the namespace and invalidate all open bindings.
  - `_RESMGR_DETACH_PATHNAME` — detach only the name from the namespace, leaving existing bindings intact. This option is useful when you’re unlinking a file or device, and you want to remove the name, but you want processes with open files to continue to use it until they close.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `resmgr_detach()` function removes pathname `id` from the pathname space of context `dpp`.
Blocking states

The `resmgr_detach()` function blocks until the `RESMGR_HANDLE_T`, that’s passed to the corresponding `resmgr_attach()`, isn’t being used in any connection function.

The effect that this has on servers is generally minimal. You should follow the following precautions to prevent potential deadlock situations:

- If you’re using the `RESMGR_HANDLE_T` as an attribute, and that attribute is locked in any of the connection callouts (i.e. open, unlink, mount, etc.), then should unlock it before calling `resmgr_detach()`. This allows any pending connection requests to complete before they’re consequently invalidated.

If you call `resmgr_detach()` from within a connection function, then the internal reference counting takes this into account and the server doesn’t deadlock.

- If two or more `resmgr_detach()` requests come in simultaneously, only one of the requests is served. The superfluous request will return with an error of -1 and `errno` set to ENOENT to indicate that the detachment process has already begun, and the entry is now invalid. If dynamically allocated, you should release `RESMGR_HANDLE_T` only after a successful return from `resmgr_detach()`.

- If `resmgr_detach()` is called and an existing client connection is established, then the I/O callout table is redirected for that client connection. The client will receive an error of EBADF when it uses the `fd` associated with that connection.

Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (<code>errno</code> is set).</td>
</tr>
</tbody>
</table>
Errors:

EINV AL  The id was never attached with resmgr_attach().

ENOENT  A previous detachment request is in progress, or the id has already been detached.

Examples:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int id;

    if ( (dpp = dispatch_create()) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \n dispatch handle." , argv[0] );
        return EXIT_FAILURE;
    }

    id = resmgr_attach ( ... );

    if ( resmgr_detach( dpp, id, 0 ) == -1 ) {
        fprintf( stderr, "Failed to remove pathname \n from the pathname space." );
        return EXIT_FAILURE;
    }
}
```

For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

Classification:

QNX Neutrino

Safety

- Cancellation point: Yes
- Interrupt handler: No

continued...
resmgr_detach()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`dispatch_create()`, `resmgr_attach()`

“Writing a Resource Manager” chapter of the Programmer’s Guide.
Resmgr_devino() — Get the device and inode number

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_devino( int id,
                  dev_t *pdevno,
                  ino64_t *pino );
```

Arguments:

- `id`: The link ID that resmgr_attach() returned.
- `pdevno`: A pointer to a dev_t object where the function can store the device number.
- `pino`: A pointer to a ino64_t object where the function can store the inode number.

Library:

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The function resmgr_devino() fills in the structures pointed to by `pdevno` and `pino` with the device number and inode number extracted from `id`.

This function is typically used to fill in:

- `iofunc_mount_t->dev`
- `iofunc_attr_t->inode`

You can use the `major()`, `minor()`, and `makedev()` macros to work with device IDs. They're defined in `<sys/types.h>` and are described in the documentation for `stat()`.
Returns:

-1 on error (errno is set); any other value on success.

Errors:

EINVAL    The id argument is invalid.

Examples:

```c
#include <sys/resmgr.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   iofunc_mount_t mount;
   iofunc_attr_t attr;

   ...
   attr.mount = &mount;
   ...
   id = resmgr_attach( ... )
   ...
   resmgr_devino(id, &mount.dev, &attr.inode);
   ...
   return EXIT_SUCCESS;
}
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

resmgr_attach(), SETIOV(), stat()
Synopsis:

```c
#include <resmgr.h>

int _resmgr_handle_grow ( unsigned min );
```

Arguments:

- `min` The number of requests that you want to accommodate.

Library:

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `_resmgr_handle_grow()` function pre-grows or allocates the resource manager database table entries to support a given number of connections to improve runtime performance by reducing the number of dynamic memory allocations.

The function pre-allocates database space for `min` requests.

Returns:

- The number of free entries in the table, or -1 if the resource manager table can’t be locked.

Classification:

- QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued…
### _resmgr_handle_grow_( )

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Synopsis:

```c
int resmgr_handle_tune(int min_handles,
                        int min_clients,
                        int max_client_handles,
                        int *min_handles_old,
                        int *min_clients_old,
                        int *max_client_handles_old);
```

Arguments:

`min_handles`  
To perform the described mapping, the resource manager framework makes use of `resmgr_handle_entry` structures. This value describes the minimum number of these structures to keep around. If more than this number are in use, they may be returned to the heap via `free()` as they’re released.

`min_clients, max_client_handles`  
To perform the described mapping, the resource manager framework makes use of hash buckets, one per client. The `min_clients` describes the minimum number of these buckets to keep around. If more than this number of clients are in communication with your resource manager, these buckets may be released back to the heap via `free()` as particular clients close all their `fds` to your manager.

The `max_client_handles` describes the size of each of these hash buckets. The maximum number of lookups to find a particular `fd`-to-OCB mapping is the client’s `max fd` divided by `max_client_handles` rounded to the nearest integer, i.e. in pseudocode:

```
ceil(max_fd/max_client_handles).
```

If this value changes, the new value takes effect for newly connected clients. Existing clients are unaffected.
resmgr_handle_tune()

If negative values are specified to any of the above three parameters, their current values are left unchanged.

* _old

If any of these are non-NULL, the corresponding value in use by the resource manager layer at the time of the call is returned.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

One of the functions of the resource manager framework is to perform the mapping of client file descriptors to structures local to the resource manager that describe these descriptors. These structures are often Open Control Blocks (OCBs). For details on OCBs, see resmgr_open_bind(). The resmgr_handle_tune() function can be used to tune certain aspects of this mapping and subsequent lookups of a client’s OCBs.

Returns:

0.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

continued . . .
resmgr_handle_tune()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

resmgr_open_bind()

Writing a Resource Manager chapter of the Programmer’s Guide
resmgr_handler()  © 2007, QNX Software Systems GmbH & Co. KG.
Handle resource manager messages

Synopsis:
#include <sys/iofunc.h>
#include <sys/dispatch.h>

    int resmgr_handler( resmgr_context_t * ctp );

Arguments:

    ctp A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.

Library:

    libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

    The resmgr_handler() function handles the message received in ctp. This function handles different I/O messages through the resource manager framework.

    The resmgr_handler() function is a special case of dispatch_handler(). You should use it only when writing a simple resource manager i.e. where there’s no need to attach pulses or messages.

Returns:

    0 Success.
    -1 An error occurred.
Examples:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    dispatch_t *dpp;
    resmgr_context_t *ctp;

    if ((dpp = dispatch_create()) == NULL) {
        fprintf( stderr, "Unable to allocate dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }

    ctp = resmgr_context_alloc( dpp );

    while (1) {
        if (( ctp = resmgr_block( ctp )) == NULL ) {
            fprintf( stderr, "block error\n" );
            return EXIT_FAILURE;
        }
        resmgr_handler( ctp );
    }
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

If you attach messages or pulses to dpp by calling message_attach(), pulse_attach(), or selectAttach(), those events aren’t dispatched by resmgr_handler(). Instead, you should call dispatch_handler().

See also:

dispatch_handler(), resmgr_attach(), resmgr_block()

“Components of a Resource Manager” in the Writing a Resource Manager chapter of the Programmer’s Guide
resmgr io func() Retrieve an I/O function from an I/O function table

Synopsis:
#include <sys/resmgr.h>

_resmgr_func_t *resmgr_io_func(
    const resmgr_io_funcs_t *funcs,
    unsigned type);

Arguments:
funcs A pointer to the resmgr_io_funcs_t structure for the table of I/O functions.
type The type of I/O function that you want to get from the table. This argument should be one of the values defined in <sys/iomsg.h>, such as _IO_READ or _IO_WRITE.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The _resmgr_io_func() function retrieves the I/O function associated with type from the function table defined by funcs.

Returns:
A pointer to the function responsible for servicing type, or NULL if the function can’t be found.

Classification:
QNX Neutrino
**resmgr_io_func()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`resmgr_io_funcs_t`, `resmgr_iolfuncs()`
resmgr_io_funcs_t

Table of POSIX-level I/O functions

Synopsis:

```c
#include <sys/resmgr.h>

typedef struct _resmgr_io_funcs {
    unsigned nfuncs;
    int (*read) (resmgr_context_t *ctp, io_read_t *msg,
                 RESMGR_OCB_T *ocb);
    int (*write) (resmgr_context_t *ctp, io_write_t *msg,
                  RESMGR_OCB_T *ocb);
    int (*close_ocb) (resmgr_context_t *ctp, void *reserved,
                      RESMGR_OCB_T *ocb);
    int (*stat) (resmgr_context_t *ctp, io_stat_t *msg,
                 RESMGR_OCB_T *ocb);
    int (*notify) (resmgr_context_t *ctp, io_notify_t *msg,
                   RESMGR_OCB_T *ocb);
    int (*devctl) (resmgr_context_t *ctp, io_devctl_t *msg,
                   RESMGR_OCB_T *ocb);
    int (*unblock) (resmgr_context_t *ctp, io_pulse_t *msg,
                    RESMGR_OCB_T *ocb);
    int (*pathconf) (resmgr_context_t *ctp, io_pathconf_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*lseek) (resmgr_context_t *ctp, io_lseek_t *msg,
                  RESMGR_OCB_T *ocb);
    int (*chmod) (resmgr_context_t *ctp, io_chmod_t *msg,
                  RESMGR_OCB_T *ocb);
    int (*chown) (resmgr_context_t *ctp, io_chown_t *msg,
                  RESMGR_OCB_T *ocb);
    int (*utime) (resmgr_context_t *ctp, io_utime_t *msg,
                  RESMGR_OCB_T *ocb);
    int (*fdopen) (resmgr_context_t *ctp, io_fdopen_t *msg,
                   RESMGR_OCB_T *ocb);
    int (*fdinfo) (resmgr_context_t *ctp, io_fdinfo_t *msg,
                   RESMGR_OCB_T *ocb);
    int (*lock) (resmgr_context_t *ctp, io_lock_t *msg,
                 RESMGR_OCB_T *ocb);
    int (*space) (resmgr_context_t *ctp, io_space_t *msg,
                  RESMGR_OCB_T *ocb);
    int (*shutdown) (resmgr_context_t *ctp, io_shutdown_t *msg,
                     RESMGR_OCB_T *ocb);
    int (*mmap) (resmgr_context_t *ctp, io_mmap_t *msg,
                 RESMGR_OCB_T *ocb);
    int (*msg) (resmgr_context_t *ctp, io_msg_t *msg,
```
resmgr_io_funcs_t

int (*reserved) (resmgr_context_t *ctp, void *msg, RESMGR_OCB_T *ocb);
int (*dup) (resmgr_context_t *ctp, io_dup_t *msg, RESMGR_OCB_T *ocb);
int (*close_dup) (resmgr_context_t *ctp, io_close_t *msg, RESMGR_OCB_T *ocb);
int (*lock_ocb) (resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb);
int (*unlock_ocb)(resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocb);
int (*sync) (resmgr_context_t *ctp, io_sync_t *msg, RESMGR_OCB_T *ocb);
}

Description:

The resmgr_connect_funcs_t structure is a table of the POSIX-level I/O functions that are used by a resource manager. You can initialize this table by calling iofunc_func_init() and then overriding the defaults with your own functions.

This structure includes nfuncs, which indicates how many functions are in the table (in case the structure grows in the future), along with these functions:

<table>
<thead>
<tr>
<th>Member</th>
<th>Used to:</th>
<th>Default:</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>Handle _IO_READ messages</td>
<td>iofunc_read_default()</td>
</tr>
<tr>
<td>write</td>
<td>Handle _IO_WRITE messages</td>
<td>iofunc_write_default()</td>
</tr>
<tr>
<td>close_ocb</td>
<td>Return the memory allocated for an OCB</td>
<td>iofunc_close_ocb_default()</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Member:</th>
<th>Used to:</th>
<th>Default:</th>
</tr>
</thead>
<tbody>
<tr>
<td>stat</td>
<td>Handle _IO_STAT messages</td>
<td>iofunc_stat_default()</td>
</tr>
<tr>
<td>notify</td>
<td>Handle _IO_NOTIFY messages</td>
<td>None</td>
</tr>
<tr>
<td>devctl</td>
<td>Handle _IO_DEVCTL messages</td>
<td>iofunc_devctl_default()</td>
</tr>
<tr>
<td>unblock</td>
<td>Unblock the resource if an operation is aborted</td>
<td>iofunc_unblock_default()</td>
</tr>
<tr>
<td>pathconf</td>
<td>Handle _IO_PATHCONF messages</td>
<td>iofunc_pathconf_default()</td>
</tr>
<tr>
<td>lseek</td>
<td>Handle _IO_LSEEK messages</td>
<td>iofunc_lseek_default()</td>
</tr>
<tr>
<td>chmod</td>
<td>Handle _IO_CHMOD messages</td>
<td>iofunc_chmod_default()</td>
</tr>
<tr>
<td>chown</td>
<td>Handle _IO_CHOWN messages</td>
<td>iofunc_chown_default()</td>
</tr>
<tr>
<td>utime</td>
<td>Handle _IO_UTIME messages</td>
<td>iofunc_utime_default()</td>
</tr>
<tr>
<td>fdopen</td>
<td>Handle _IO_OPENFD messages</td>
<td>iofunc_openfd_default()</td>
</tr>
<tr>
<td>fdinfo</td>
<td>Handle _IO_FDINFO messages</td>
<td>iofunc_fdinfo_default()</td>
</tr>
<tr>
<td>lock</td>
<td>Handle _IO_LOCK messages</td>
<td>iofunc_lock_default()</td>
</tr>
<tr>
<td>space</td>
<td>Allocate or free memory for the resource.</td>
<td>None</td>
</tr>
<tr>
<td>shutdown</td>
<td>Reserved</td>
<td>None</td>
</tr>
</tbody>
</table>

continued...
### resmgr_io_funcs_t

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Member</th>
<th>Used to:</th>
<th>Default:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmap</td>
<td>Handle _IO_MMAP messages</td>
<td>iofunc_mmap_default()</td>
</tr>
<tr>
<td>msg</td>
<td>Handle a more general messaging scheme to control the device</td>
<td>None</td>
</tr>
<tr>
<td>reserved</td>
<td>Not applicable</td>
<td>None</td>
</tr>
<tr>
<td>dup</td>
<td>Handle _IO_DUP messages</td>
<td>None — handled by the base layer</td>
</tr>
<tr>
<td>close_dup</td>
<td>Handle _IO_CLOSE messages</td>
<td>iofunc_close_dup_default()</td>
</tr>
<tr>
<td>lock_ocb</td>
<td>Lock the attributes for a group of messages</td>
<td>iofunc_lock_ocb_default()</td>
</tr>
<tr>
<td>unlock_ocb</td>
<td>Unlock the attributes for a group of messages</td>
<td>iofunc_unlock_ocb_default()</td>
</tr>
<tr>
<td>sync</td>
<td>Handle _IO_SYNC messages</td>
<td>iofunc_sync_default()</td>
</tr>
</tbody>
</table>

If you use iofunc_lock_default(), you must also use iofunc_close_dup_default() and iofunc_unblock_default().

### Classification:

QNX Neutrino

### See also:

iofunc_chmod_default(), iofunc_chown_default(),
iofunc_close_dup_default(), iofunc_close_ocb_default(),
iofunc_devctl_default(), iofunc_fdinfo_default(), iofunc_func_init(),
iofunc_lock_default(), iofunc_lock_ocb_default(),
iofunc_lseek_default(), iofunc_mmap_default(),
iofunc_openfd_default(), iofunc_pathconf_default(),
resmgr_io_funcs_t

iofunc_read_default(), iofunc_stat_default(), iofunc_sync_default(),
iofunc_unblock_default(), iofunc_unlock_ocb_default(),
iofunc_utime_default(), iofunc_write_default(),
resmgr_connect_funcs_t

Writing a Resource Manager chapter of the QNX Neutrino
Programmer’s Guide
**resmgr_iofuncs()**  © 2007, QNX Software Systems GmbH & Co. KG.

Extract the I/O function pointers associated with client connections

**Synopsis:**
```
#include <resmgr.h>

const resmgr_io_funcs_t * resmgr_iofuncs(
    resmgr_context_t * ctp;
    struct _msg_info * info);
```

**Arguments:**
- **ctp** A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **info** A pointer to the `_msg_info` structure that describes the binding to the client. You should fill this structure by calling `MsgInfo()`.

**Library:**
- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `resmgr_iofuncs()` function retrieves the I/O function callout table associated with the client connections described by binding specified by `info`.

Note that context information pointed to by `ctp` actually contains `info`.

**Returns:**
- A pointer to the `resmgr_io_funcs_t` I/O function callout table, or NULL if the binding can’t be found or an error occurs.
resmgr_iofuncs()

Errors:

ESRCH The connection can’t be located in the resource manager’s table.
ENOMEM There is no memory available for the operation.
EINV AL Invalid arguments were used.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

_msg_info, MsgInfo(), resmgr_io_funcs_t, _resmgr_ocb(), resmgr_open_bind(), resmgr_unbind()
**resmgr_msgread()**

Read a message from a client

**Synopsis:**

```c
#include <sys/resmgr.h>

int resmgr_msgread( resmgr_context_t * ctp,
                    void * msg,
                    int size,
                    int offset );
```

**Arguments:**

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.
- `msg` A pointer to a buffer where the function can store the data.
- `bytes` The number of bytes that you want to read. These functions don’t let you read past the end of the thread’s message; they return the number of bytes actually read.
- `offset` An offset into the thread’s send message that indicates where you want to start reading the data.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `resmgr_msgread()` function is a convenience function that you should in a resource manager instead of `MsgRead()`.

You’ll use `resmgr_msgread()` when you handle combine messages, where the offset of the rest of the message that’s to be read is additionally offset by previous combine message elements. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the *Programmer’s Guide.*
resmgr_msgread()  

Returns:

The same values as MsgRead(): the number of bytes read, or -1 if an error occurs (errno is set).

Errors:

- EFAULT: A fault occurred in a server’s address space when it tried to access the caller’s message buffers.
- ESRCH: The thread indicated by ctp -> rcvid doesn’t exist or its connection is detached.
- ESRVRFault: A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

MsgRead(), resmgr_context_t, resmgr_msgreadv(), resmgr_msgwrite(), resmgr_msgwritev()

“Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide
resmgr_msgreadv()

Read a message from a client

Synopsis:

```c
#include <sys/resmgr.h>

int resmgr_msgreadv( resmgr_context_t * ctp,
                      iov_t * rmsg,
                      int rparts,
                      int offset );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.
- `riov` An array of buffers where the functions can store the data.
- `rparts` The number of elements in the `riov` array.
- `offset` An offset into the thread’s send message that indicates where you want to start reading the data.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This `resmgr_msgreadv()` function is a convenience function that you should use in a resource manager instead of `MsgReadv()`.

You’ll use `resmgr_msgreadv()` when handling combine messages, where the offset of the rest of the message that is to be read is additionally offset by previous combine message elements. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.
resmgr_msgreadv()

Returns:

The same values as MsgReadv(): the number of bytes read, or -1 if an error occurs (errno is set).

Errors:

- **EFAULT**: A fault occurred in a server’s address space when it tried to access the caller’s message buffers.
- **ESRCH**: The thread indicated by ctp -> rcvid doesn’t exist or has had its connection detached.
- **ESRVRFAULT**: A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

MsgReadv(), resmgr_context_t, resmgr_msgreadv(), resmgr_msgwrite(), resmgr_msgwritev()

“Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide
resmgr_msgwrite()

Write a message to a client

Synopsis:

```
#include <sys/resmgr.h>

int resmgr_msgwrite( resmgr_context_t *ctp,
                     const void *msg,
                     int size,
                     int offset );
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.
- `msg` A pointer to a buffer that contains the data you want to write.
- `size` The number of bytes that you want to write. This function doesn’t let you write past the end of the sender’s buffer; it returns the number of bytes actually written.
- `offset` An offset into the sender’s buffer that indicates where you want to start writing the data.

Library:

`libc`

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `resmgr_msgwrite()` is a cover for `MsgWrite()` and performs the exact same functionality.
resmgr_msgwrite()

Returns:

The same values as MsgWrite(); the number of bytes written, or -1 if an error occurs (errno is set).

Errors:

EFAULT A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffer.

ESRCH The thread indicated by ctp -> rcvid does not exist or has had its connection detached.

ESRVFAULT A fault occurred when the kernel tried to access the buffers provided.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

MsgWrite(), resmgr_context_t, resmgr_msgread(), resmgr_msgreadv(), resmgr_msgwritev()

“Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide
resmgr_msgwritev()
Write a message to a client

**Synopsis:**

```c
#include <sys/resmgr.h>

int resmgr_msgwritev( resmgr_context_t *ctp,
    const iov_t *smsg,
    int sparts,
    int offset );
```

**Arguments:**

- **ctp**  
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions. This function extracts the rcvid from this structure.

- **iov**  
  An array of buffers that contains the data you want to write.

- **parts**  
  The number of elements in the array. This function doesn’t let you write past the end of the sender’s buffer; it returns the number of bytes actually written.

- **offset**  
  An offset into the sender’s buffer that indicates where you want to start writing the data.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `resmgr_msgwritev()` function is a cover function for `MsgWritev()`, and performs the exact same functionality. It’s provided for consistency with `resmgr_msgwrite()`.
resmgr_msgwritev()

Returns:
The number of bytes written, or -1 if an error occurred (errno is set).

Errors:

EFAULT A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffer.

ESRCH The thread indicated by ctp -> rcvid does not exist or has had its connection detached.

ESRVRFAULT A fault occurred when the kernel tried to access the buffers provided.

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

MsgWritev(), resmgr_context_t, resmgr_msgread(), resmgr_msgreadv(), resmgr_msgwrite()  

“Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide
Get a given number of parts from the ctp->iov structure

**Synopsis:**

```c
#include <sys/resmgr.h>

_RESMGR_NPARTS( int num )
```

**Arguments:**

`num` The number of parts that you want to get.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The macro `__RESMGR_NPARTS()` indicates to the caller to get `num` parts from the `ctp->iov` structure (see `resmgr_context_t`). The macro is similar to:

```
MsgReply( ctp->rcvid, ctp->status, ctp->iov, num ).
```

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The `resmgr_attach()` function should set `attr->nparts_max` to be the maximum value for `num`.

See also:

MsgReply(), resmgr_attach(), resmgr_context_t, _RESMGR_PTR(), _RESMGR_STATUS()
**resmgr_ocb()**

Retrieve an Open Control Block

### Synopsis:

```c
#include <sys/resmgr.h>

void * _resmgr_ocb( resmgr_context_t * ctp, 
                   struct _msg_info * info);
```

### Arguments:

- **ctp**
  - A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
  - Note that `ctp` actually contains `info`.

- **info**
  - A pointer to a `msg_info` structure that indicates which client mapping you want to retrieve information about. To fill in the structure, call `MsgInfo()`.

### Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:

The `_resmgr_ocb()` function queries the internal resource manager database, which maps client connections to the server Open Control Block (OCB), to retrieve the OCB pointer that was previously bound using `resmgr_open_bind()`.

### Returns:

- A pointer to the OCB for the matching binding, or NULL if the binding can’t be found or an error occurred.

- The OCB can be a structure that you define. By default, it’s of type `iofunc_ocb_t`.

---

2588   C Library — P to R   September 10, 2007
Errors:

ESRCH The connection can’t be located in the resource manager table.
ENOMEM There isn’t enough memory available for the operation.
EINVAL Invalid arguments were used.

Classification:

QNX Neutrino

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

See also:

iofunc_ocb_t, _msg_info, MsgInfo(), resmgr_iofuncs(), resmgr_open_bind(), resmgr_unbind()
**resmgr_open_bind()**

Associate an OCB with an open request

**Synopsis:**

```c
#include <sys/resmgr.h>

int resmgr_open_bind(
    resmgr_context_t* ctp,
    void* ocb,
    const resmgr_io_funcs_t* iofuncs );
```

**Arguments:**

- **ctp**
  A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

- **ocb**
  A pointer to the Open Control Block that you want to bind to the open request. An OCB is usually a structure of type `iofunc_ocb_t`, but you can define your own.

- **iofuncs**
  A pointer to the `resmgr_io_funcs_t` structure that defines the I/O functions for the resource manager.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `resmgr_open_bind()` function is the lowest-level call in the resource manager library used for handling open messages. It associates the Open Control Block (OCB) with a process identified by the `id` and `info` members of `ctp`. 
You must use this function as part of the handling of an _IO_OPEN message. In practice, you don’t call this function directly; you typically use either iofunc_open_default() or iofunc_ocb_attach(). (The iofunc_open_default() function calls iofunc_ocb_attach(), which in turn calls resmgr_open_bind()).

An internal data structure is allocated that maintains the number of links to the OCB. On a file descriptor dup(), the link count is incremented and on a close() it’s decremented. When the count reaches zero, the close_ocb() callout specified in io_funcs is called.

In the most general case, the OCB is an arbitrary structure that you define that can hold information describing an open file, or just a simple int to hold the open mode for checking in the read() and write() callouts.

In the typical case, however, the OCB is a structure that contains at least the members as defined by the typedef iofunc_ocb_t. This typedef defines a common OCB structure that can then be used by the POSIX layer helper functions (all functions beginning with the name iofunc_*) etc. The advantage of this approach is that your resource manager gets POSIX behavior for free, without any additional work on your part.

The attr argument to the open() callout is also typically saved in the OCB. The well defined iofunc_ocb_t has a member called attr to which you must assign the value of the attr argument. This lets the POSIX helper functions access information about the current open session (as stored in the OCB) as well as information about the device itself (as stored in the attributes structure, ocb -> attr).

For a detailed discussion, including several examples, see the Writing a Resource Manager chapter of the Programmer’s Guide.

**Returns:**

0 Success.

-1 An error occurred (errno is set).
resmgr_open_bind()

Errors:

- EINVAL: The id and/or info members of ctp aren’t valid.
- ENOMEM: Insufficient memory to allocate an internal data structure.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

iofunc_ocb_attach(), iofunc_ocb_t, iofunc_open_default(), resmgr_context_t, resmgr_io_funcs_t, resmgr_unbind()

Writing a Resource Manager chapter of the Programmer’s Guide
resmgr_pathname()

Return the pathname associated with an ID

Synopsis:

```c
#include <sys/resmgr.h>

int resmgr_pathname( int id,
                     unsigned flags,
                     char* path,
                     int maxbuf );
```

Arguments:

- **id**: The link ID that `resmgr_attach()` returned.
- **flags**: Flags that affect the operation:
  - `_RESMGR_PATHNAME_LOCALPATH` — get a shortened pathname that’s usable only on the local machine. By default, this function gets a globally unique pathname.
- **path**: A pointer to a buffer where the function can store the pathname.
- **maxbuf**: The size of the buffer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `resmgr_pathname()` function returns the pathname associated with an `id` that’s returned from `resmgr_attach()`, it’s also the `ctp->id` value of all the resmgr callout functions.

If the `id` was obtained from calling `resmgr_attach()` with `_RESMGR_FLAG_DIR` specified, then the path name includes a trailing slash.

By default, this function calls:
resmgr_pathname()

netmgr_nttostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)
If you specify _RESMGR_PATHNAME_LOCALPATH, it calls
netmgr_nttostr(ND2S_DIR_SHOW|ND2S_LOCAL_STR, nd, buf, sizeofbuf)
to return a shortened path that’s usable on your local node only. This
is useful for display.

Returns:
The length of the path, including the terminating NULL character, or
-1 if an error occurs (errno is set).

Errors:
EFAULT A fault occurred in a server’s address space when
it tried to access the caller’s message buffers.
ESRCH The thread indicated by ctp -> rcvid doesn’t exist
or its connection is detached.
ESRVRFAULT A fault occurred when the kernel tried to access
the buffers provided.

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

netmgr_ndiostr(), resmgr_attach()
Synopsis:

```c
#include <sys/resmgr.h>

_RESMGR_PTR( resmgr_context_t ctp,
    void msg,
    size_t nbytes )
```

Arguments:

- **ctp**: A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.
- **msg**: The value you want to use for the structure’s `iov_base` member.
- **nbytes**: The value you want to use for the structure’s `iov_len` member.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_RESMGR_PTR()` macro gets one part from the `ctp->iov` structure (see `resmgr_context_t`) and fills in its fields. The macro is equivalent to:

```c
SETIOV (ctp->iov, msg, nbytes)
```

returning `_RESMGR_NPARTS (1)`.

Classification:

QNX Neutrino
Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

resmgr_context_t, _RESMGR_NPARTS(), _RESMGR_STATUS(), SETIOV()
RESMGR_STATUS() © 2007, QNX Software Systems GmbH & Co. KG.
Set the status member of a resource-manager context

Synopsis:
#include <sys/resmgr.h>

_RESMGR_STATUS( resmgr_context_t *ctp,
                int status )

Arguments:
ctp A pointer to a resmgr_context_t structure that the resource-manager library uses to pass context information between functions.
status The status that you want to set.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The _RESMGR_STATUS() macro sets the status member in the resmgr_context_t structure.
The resource manager library uses this status when it returns the value from _RESMGR_NPARTS() for an I/O or connect function, such as:
MsgReply ( ctp->rcvid, ctp->status, ctp->iov, num ).

Classification:
QNX Neutrino

Safety
Cancellation point No
Interrupt handler Yes
continued...
_RESMGR_STATUS()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

resmgr_connect_funcs_t, resmgr_io_funcs_t, _RESMGR_NPARTS(), _RESMGR_PTR()
Disassociate an OCB from an open request

Synopsis:

```c
#include <sys/resmgr.h>

int resmgr_unbind( resmgr_context_t * ctp);
```

Arguments:

- `ctp` A pointer to a `resmgr_context_t` structure that the resource-manager library uses to pass context information between functions.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `resmgr_unbind()` function removes a binding in the internal resources manager database (which maps client connections to server OCB pointers). The binding must previously have been bound by `resmgr_open_bind()`.

The binding is reference counted; if multiple connections are established with the same binding, the binding is freed only when the last connection is removed.

You should use `MsgInfo()` to fill the `info` structure in `ctp` with information about which client mapping to retrieve.

Returns:

- `-1` Failure.
- `0` Success.
Errors:

ESRCH The binding can’t be located in the resource manager table.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

MsgInfo(), resmgr_context_t, resmgr_open_bind()

Writing a Resource Manager chapter of the Programmer’s Guide
**rewind()**

*Rewind a file stream to the beginning of the file*

**Synopsis:**

```c
#include <stdio.h>

void rewind( FILE *fp );
```

**Arguments:**

- `fp` The file stream that you want to rewind.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `rewind()` function rewinds the file stream specified by `fp` to the beginning of the file. It’s equivalent to calling `fseek()` like this:

```c
fseek( fp, 0L, SEEK_SET );
```

except that the error indicator for the stream is cleared.

**Examples:**

This example shows how you might implement a two-pass assembler:

```c
#include <stdio.h>
#include <stdlib.h>

void assemble_pass( FILE *fp, int passno )
{
    printf( "Pass %d\n", passno );
    /* Do more work on the fp */
    switch( passno ) {
    case 1:
        /* do the first-pass work */
        break;

    case 2:
        /* do the second-pass work */
        
```
rewind()

```c
break;

default:
    break;
}

int main( void )
{
    FILE *fp;

    fp = fopen( "program.s", "r" );
    if( fp != NULL ) {
        assemble_pass( fp, 1 );
        rewind( fp );

        assemble_pass( fp, 2 );
        fclose( fp );

        return EXIT_SUCCESS;
    }

    puts( "Error opening program.s" );

    return EXIT_FAILURE;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

clearerr(), fopen(), fseek()
### Synopsis:
```c
#include <sys/types.h>
#include <dirent.h>

void rewinddir( DIR * dirp );
```

### Arguments:
- `dirp`  A pointer to the directory stream of the directory to rewind.

### Library:
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

### Description:
The `rewinddir()` function rewinds the directory stream specified by `dirp` to the start of the directory. The directory stream will now refer to the current state of the directory, as if the calling thread had called `opendir()` again.

The result of using a directory stream after one of the `exec*()` or `spawn*()` family of functions is undefined. After a call to `fork()`, either the parent or the child (but not both) can continue processing the directory stream, using the `readdir()` and `rewinddir()` functions. If both the parent and child processes use these functions, the result is undefined. Either (or both) processes may use `closedir()`.

### Examples:
List all the files in a directory, create a new file, and then list the directory contents again:
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```
#include <dirent.h>
#include <stdlib.h>

int main( void )
{
    DIR *dirp;
    struct dirent *direntp;
    int filedes;

    dirp = opendir( "'/home/fred'" );
    if( dirp != NULL ) {
        printf( "Old directory listing\n" );
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;
            printf( "%s\n", direntp->d_name );
        }

        filedes = creat( "'/home/fred/file.new'",
                         S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
        close( filedes );

        rewinddir( dirp );
        printf( "New directory listing\n" );
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;
            printf( "%s\n", direntp->d_name );
        }
        closedir( dirp );
    }

    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

Cancellation point Yes
Interrupt handler No
Signal handler Yes

continued…
rewinddir()

Safety

Thread: Yes

See also:

closedir(), opendir(), readdir(), readdir_r(), seekdir()
Rgetsockname()

Get the name of a socket (via a SOCKS server)

Synopsis:

```c
#include <sys/socket.h>

int Rgetsockname( int s,
                    struct sockaddr * name,
                    int * namelen );
```

Arguments:

- `s` The file descriptor of the socket whose name you want to get.
- `name` A pointer to a `sockaddr` object where the function can store the socket’s name.
- `namelen` A pointer to a `socklen_t` object that initially indicates the amount of space pointed to by `name`. The function updates `namelen` to contain the actual size of the name (in bytes).

Library:

```
libsocks
```

Use the `-l socks` option to `qcc` to link against this library.

Description:

The `Rgetsockname()` function is a cover function for `getsockname()` — the difference is that `Rgetsockname()` does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.

Returns:

- `0` Success.
- `-1` An error occurred (`errno` is set).
Rgetsockname()

Classification:

SOCKS

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

getsockname(), Raccept(), Rbind(), Rconnect(), Rlisten(), Rcmd(), Rselect(), SOCKSinit()

SOCKS — A Basic Firewall


**rindex()**

Find the last occurrence of a character in a string

**Synopsis:**

```
#include <strings.h>

char *rindex( const char *s, 
               int c );
```

**Arguments:**

- **s**  The string you want to search. This string must end with a null (\0) character. The null character is considered to be part of the string.
- **c**  The character you’re looking for.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `rindex()` function returns a pointer to the last occurrence of the character `c` in the string `s`.

**Returns:**

A pointer to the character, or NULL if the character doesn’t occur in the string.

**Classification:**

POSIX 1003.1 XSI

**Safety**

Cancellation point  No

*continued.*
rindex()

Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`index()`, `strchr()`, `strrchr()`
r

rint(), rintf()

Round to the nearest integral value

Synopsis:

```c
#include <math.h>

double rint ( double x );

float rintf ( float x );
```

Arguments:

x The number that you want to round.

Library:

libm

Use the -l m option to qcc to link against this library.

Description:

The rint() and rintf() functions return the integral value nearest \( x \) in the direction of the current rounding mode.

If the current rounding mode rounds toward negative infinity, then rint() is identical to floor(). If the current rounding mode rounds toward positive infinity, then rint() is identical to ceil().

Returns:

An integer (represented as a double precision number) nearest \( x \) in the direction of the current rounding mode (IEEE754).

<table>
<thead>
<tr>
<th>If ( x ) is:</th>
<th>rint() returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>±Infinity</td>
<td>( x )</td>
</tr>
<tr>
<td>NAN</td>
<td>NAN</td>
</tr>
</tbody>
</table>
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b;
    a = 0.7;
    b = rint(a);
    printf("Round Native mode %f -> %f \n", a, b);

    return(0);
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`ceil()`, `floor()`
Synopsis:

```c
#include <sys/socket.h>

int Rlisten( int s,
            int backlog );
```

Arguments:

- `s` The descriptor for the socket that you want to listen on. You can create a socket by calling `socket()`.
- `backlog` The maximum length that the queue of pending connections may grow to.

Library:

- `libsocks`

Use the `-l socks` option to `qcc` to link against this library.

Description:

The `Rlisten()` function is a cover function for `listen()` — the difference is that `Rlisten()` does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.

Returns:

- `0` Success.
- `-1` An error occurred (`errno` is set).

Classification:

- SOCKS
Rlisten()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

listen(), Raccept(), Rbind(), Rconnect(), Rgetsockname(), Rcmd(), Rselect(), SOCKSinit()

SOCKS — A Basic Firewall
Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>

int rmdir( const char* path );
```

Arguments:

- `path` The path of the directory that you want to delete. This path can be relative to the current working directory, or an absolute path.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `rmdir()` function removes (deletes) the specified directory. The directory must not contain any files or directories.

If the directory is the current working directory of any process, `rmdir()` returns -1 and sets `errno` to EINVAL. If the directory is the root directory, the effect of this function depends on the filesystem.

The space occupied by the directory is freed, making it inaccessible, if its link count becomes zero and no process has the directory open (`opendir()`). If a process has the directory open when the last link is removed, the . and .. entries are removed and no new entries can be created in the directory. In this case, the directory will be removed when all references to it have been closed (`closedir()`).

When successful, `rmdir()` marks `st_cftime` and `st_mtime` for update in the parent directory.
**rmdir()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

- **0** Success.
- **-1** An error occurred (*errno* is set).

**Errors:**

- **EACCES** Search permission is denied for a component of *path*, or write permission is denied on the parent directory of the directory to be removed.
- **EBUSY** The directory named by *path* can’t be removed because it’s being used by another process, and the implementation considers this to be an error.
- **EEXIST** The *path* argument names a directory that isn’t empty.
- **ELOOP** Too many levels of symbolic links.
- **ENAMETOOLONG** The argument *path* exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.
- **ENOENT** The specified *path* doesn’t exist, or *path* is an empty string.
- **ENOSYS** The *rmdir()* function isn’t implemented for the filesystem specified in *path*.
- **ENOTDIR** A component of *path* isn’t a directory.
- **ENOTEMPTY** The *path* argument names a directory that isn’t empty.
- **EROFS** The directory entry to be removed resides on a read-only filesystem.
**Examples:**

To remove the directory called `/home/terry`:

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void )
{
    (void)rmdir( "/home/terry" );

    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`chdir()`, `chmod()`, `errno`, `getcwd()`, `mkdir()`, `stat()`
System packet forwarding database

Synopsis:
```
#include <sys/socket.h>
#include <net/if.h>
#include <net/route.h>

int socket( PF_ROUTE,
            SOCK_RAW,
            int family );
```

Description:
QNX TCP/IP provides some packet routing facilities.

The following information applies only to the full TCP/IP stack. For information on how the tiny TCP/IP stack can change the routing table, see npm-ttcpip.so.

The socket manager maintains a routing information database that’s used in selecting the appropriate network interface when transmitting packets.

A user process (or possibly multiple cooperating processes) maintains this database by sending messages over a special kind of socket. This supplants fixed-size ioctl()s used in earlier releases. Routing table changes may be carried out only by the superuser.

This interface may spontaneously emit routing messages in response to external events, such as receipt of a redirect or of a failure to locate a suitable route for a request. The message types are described in greater detail below.

Routing database entries
Routing database entries come in two flavors: for a specific host or for all hosts on a generic subnetwork (as specified by a bit mask and value under the mask). The effect of wildcard or default routing may be achieved by using a mask of all zeros. There may be hierarchical routes.
When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it’s ready for traffic. Normally the protocol specifies the route through each interface as a “direct” connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (i.e. the packet is forwarded).

Routing packets

When routing a packet, the kernel attempts to find the most specific route matching the destination. (If there are two different mask and value-under-the-mask pairs that match, the more specific is the one with more bits in the mask. A route to a host is regarded as being supplied with a mask of as many ones as there are bits in the destination). If no entry is found, the destination is declared to be unreachable, and a routing-miss message is generated if there are any listeners on the routing control socket described below.

A wildcard routing entry is specified with a zero destination address value and a mask of all zeroes. Wildcard routes are used when the system fails to find other routes matching the destination. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

Routing control messages

To open the channel for passing routing control messages, use the socket call shown in the synopsis above.

The family parameter may be AF_UNSPEC, which provides routing information for all address families, or it can be restricted to a specific address family. There can be more than one routing socket open per system.

Messages are formed by a header followed by a small number of sockaddr (with variable length) interpreted by position and delimited by the new length entry in the sockaddr. An example of a
message with four addresses might be a redirect: Destination, Netmask, Gateway, and Author of the redirect. The interpretation of which addresses are present is given by a bit mask within the header; the sequence is least-significant to most-significant bit within the vector.

Any messages sent to the socket manager are returned, and copies are sent to all interested listeners. The interface provides the process ID for the sender. To distinguish between outstanding messages, the sender may use an additional sequence field. However, message replies may be lost when socket manager buffers are exhausted.

The interface may reject certain messages, as indicated in the \texttt{rtm_errno} field.

<table>
<thead>
<tr>
<th>This error occurs:</th>
<th>If:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEXIST</td>
<td>Requested to duplicate an existing entry.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>Requested to delete a nonexistent entry.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient resources were available to install a new route.</td>
</tr>
</tbody>
</table>

In the current implementation, all routing processes run locally, and the values for \texttt{rtm_errno} are available through the normal \texttt{errno} mechanism, even if the routing reply message is lost.

A process may avoid the expense of reading replies to its own messages by calling \texttt{setsockopt()}, to turn off the SO\_USELOOPBACK option at the SOL\_SOCKET level. A process may ignore all messages from the routing socket by doing a \texttt{shutdown()} system call for further input.

If a route is in use when it’s deleted, the routing entry is marked down and removed from the routing table, but the resources associated with it won’t be reclaimed until all references to it are released. User processes can obtain information about the routing entry to a specific destination by using a RTM\_GET message or by calling \texttt{sysctl()}. 
The messages are:

- `#define RTM_ADD 0x1 /* Add Route */`
- `#define RTM_DELETE 0x2 /* Delete Route */`
- `#define RTM_CHANGE 0x3 /* Change Metrics, Flags, or Gateway */`
- `#define RTM_GET 0x4 /* Report Information */`
- `#define RTM_LOSING 0x5 /* Kernel Suspects Partitioning */`
- `#define RTM_REDIRECT 0x6 /* Told to use different route */`
- `#define RTM_MISS 0x7 /* Lookup failed on this address */`
- `#define RTM_RESOLVE 0xb /* request to resolve dst to LL addr */`
- `#define RTM_NEWADDR 0xc /* address being added to iface */`
- `#define RTM_DELADDR 0xd /* address being removed from iface */`
- `#define RTM_IFINFO 0xe /* iface going up/down etc. */`

A message header consists of one of the following:

```c
struct rt_msghdr {
    u_short rtm_msglen; /* skip over non-understood msgs */
    u_char rtm_version; /* future binary compatibility */
    u_char rtm_type; /* message type */
    u_short rtm_index; /* index for associated ifp */
    int rtm_flags; /* flags, incl kern & message, e.g. DONE */
    int rtm_addrs; /* bitmask identifying sockaddrs in msg */
    pid_t rtm_pid; /* identify sender */
    int rtm_seq; /* for sender to identify action */
    int rtm_errno; /* why failed */
    int rtm_use; /* from rtentry */
    u_long rtm_inits; /* which metrics we're initializing */
    struct rt_metrics rtm_rmx; /* metrics themselves */
};
```

```c
struct if_msghdr {
    u_short ifm_msglen; /* to skip over non-understood msgs */
    u_char ifm_version; /* future binary compatibility */
    u_char ifm_type; /* message type */
    int ifm_addr; /* like rtm_addr */
    int ifm_flags; /* value of if_flags */
    u_short ifm_index; /* index for associated ifp */
    struct if_data ifm_dat /* statistics and other data about if */
};
```

```c
struct ifa_msghdr {
    u_short ifam_msglen; /* to skip over non-understood msgs */
    u_char ifam_version; /* future binary compatibility */
    u_char ifam_type; /* message type */
    int ifam_addr; /* like rtm_addr */
    int ifam_flags; /* value of ifa_flags */
    u_short ifam_index; /* index for associated ifp */
    int ifam_metric; /* value of ifa_metric */
};
```

The RTM_IFINFO message uses an `if_msghdr` header. The RTM_NEWADDR and RTM_DELADDR messages use an `ifa_msghdr` header. All other messages use the `rt_msghdr` header.

The metrics structure is:
struct rt_metrics {
    u_long rmx_locks; /* Kernel must leave these values alone */
    u_long rmx_mtu; /* MTU for this path */
    u_long rmx_hopcount; /* max hops expected */
    u_long rmx_expire; /* lifetime for route, e.g. redirect */
    u_long rmx_recvpipe; /* inbound delay-bandwidth product */
    u_long rmx_sendpipe; /* outbound delay-bandwidth product */
    u_long rmx_ssthresh; /* outbound gateway buffer limit */
    u_long rmx_rtt; /* estimated round trip time */
    u_long rmx_rttvar; /* estimated rtt variance */
    u_long rmx_pktsent; /* packets sent using this route */
};

Flags include the values:

#define RTF_UP 0x1 /* route usable */
#define RTF_GATEWAY 0x2 /* destination is a gateway */
#define RTF_HOST 0x4 /* host entry (net otherwise) */
#define RTF_REJECT 0x8 /* host or net unreachable */
#define RTF_DYNAMIC 0x10 /* created dynamically (by redirect) */
#define RTF_MODIFIED 0x20 /* modified dynamically (by redirect) */
#define RTF_DONE 0x40 /* message confirmed */
#define RTF_MASK 0x80 /* subnet mask present */
#define RTF_CLONING 0x100 /* generate new routes on use */
#define RTF_XRESOLVE 0x200 /* external daemon resolves name */
#define RTF_LLINFO 0x400 /* generated by ARP or ESIS */
#define RTF_STATIC 0x800 /* manually added */
#define RTF_BLACKHOLE 0x1000 /* just discard pkts (during updates) */
#define RTF_PROTO2 0x4000 /* protocol specific routing flag */
#define RTF_PROTO1 0x8000 /* protocol specific routing flag */

Specifiers for metric values in rmx_locks and rtm_inits are:

#define RTV_MTU 0x1 /* init or lock _mtu */
#define RTV_HOPCOUNT 0x2 /* init or lock _hopcount */
#define RTV_EXPIRE 0x4 /* init or lock _expire */
#define RTV_RPIPE 0x8 /* init or lock _recvpipe */
#define RTV_SPIPE 0x10 /* init or lock _sendpipe */
#define RTV_SSTHRESH 0x20 /* init or lock _ssthresh */
#define RTV_RTT 0x40 /* init or lock _rtt */
#define RTV_RTTVAR 0x80 /* init or lock _rttvar */

Specifiers for which addresses are present in the messages are:

#define RTA_DST 0x1 /* destination sockaddr present */
#define RTA_GATEWAY 0x2 /* gateway sockaddr present */
#define RTA_NETMASK 0x4 /* netmask sockaddr present */
#define RTA_GENMASK 0x8 /* cloning mask sockaddr present */
#define RTA_IFP 0x10 /* interface name sockaddr present */
#define RTA_IFA 0x20 /* interface addr sockaddr present */
#define RTA_AUTHOR 0x40 /* sockaddr for author of redirect */
#define RTA_BRD 0x80 /* for NEWADDR, */
        /* broadcast or p-p dest addr */
Examples:

Use the following code to set the default route:

```c
#include <sys/socket.h>
#include <sys/uio.h>
#include <unistd.h>
#include <net/route.h>
#include <netinet/in.h>
#include <stdio.h>
#include <libgen.h>
#include <arpa/inet.h>
#include <process.h>
#include <errno.h>

struct my_rt
{
    struct rt_msghdr rt;
    struct sockaddr_in dst;
    struct sockaddr_in gate;
    struct sockaddr_in mask;
};

int main(int argc, char **argv)
{
    int s;
    struct rt_msghdr *rtm;
    struct sockaddr_in *dst, *gate, *mask;
    struct my_rt my_rt;

    if(argc < 2)
    {
        fprintf(stderr, 
            "Usage: %s: <ip_addr_of_default_gateway>
            basename(argv[0]));
        return 1;
    }

    if((s = socket(AF_ROUTE, SOCK_RAW, 0)) == -1)
    {
        perror("socket");
        return 1;
    }

    memset(&my_rt, 0x00, sizeof(my_rt));

    rtm = &my_rt.rt;

    dst = &my_rt.dst;
    gate = &my_rt.gate;
```
ROUTE

mask = &my_rt.mask;

rtm->rtm_type = RTM_ADD;
rtm->rtm_flags = RTF_UP | RTF_GATEWAY | RTF_STATIC;
rtm->rtm_msglen = sizeof(my_rt);
rtm->rtm_version = RTM_VERSION;
rtm->rtm_seq = 1234;
rtm->rtm_addrs = RTA_DST | RTA_GATEWAY | RTA_NETMASK;
rtm->rtm_pid = getpid();

dst->sin_len = sizeof(*dst);
dst->sin_family = AF_INET;

mask->sin_len = sizeof(*mask);
mask->sin_family = AF_INET;

gate->sin_len = sizeof(*gate);
gate->sin_family = AF_INET;
inet_aton(argv[1], &gate->sin_addr);

AGAIN:

  if(write(s, rtm, rtm->rtm_msglen) < 0)
  {
    if(errno == EEXIST && rtm->rtm_type == RTM_ADD)
    {
      rtm->rtm_type = RTM_CHANGE;
      goto AGAIN;
    }
    perror("write");
    return 1;
  }
  return 0;

See also:

setsockopt(), socket(), sysctl()
npm-tcpipe.so in the Utilities Reference
Rrcmd()

Execute a command on a remote host (via a SOCKS server)

Synopsis:

```c
int Rrcmd( char ** ahost,
           int inport,
           const char * locuser,
           const char * remuser,
           const char * cmd,
           int * fd2p );
```

Arguments:

- **ahost**: The name of the host that you want to execute the command on. If the function can find the host, it sets *ahost to the standard name of the host.
- **inport**: The well-known Internet port on the host, where the server resides.
- **locuser**: The user ID on the local machine.
- **remuser**: The user ID on the remote machine.
- **cmd**: The command that you want to execute.
- **fd2p**: See rcmd().

Library:

```
libsocks
```

Use the `-l socks` option to `qcc` to link against this library.

Description:

The Rrcmd() function is a cover function for rcmd() — the difference is that Rrcmd() does its job via a SOCKS server.

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.
Returns:
A valid socket descriptor; or -1 if an error occurs and a message is printed to standard error.

Classification:
SOCKS

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:
Raccept(), Rbind(), rcmd(), Rconnect(), Rgetsockname(), Rlisten(), Rselect(), SOCKSinit()
SOCKS — A Basic Firewall
Synopsis:

```c
#include <unistd.h>

int rresvport( int * port );
```

Arguments:

`port` An address in the privileged port space. Privileged Internet ports are those in the range 0 to 1023. Only the superuser may bind this type of address to a socket.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `rresvport()` function returns a descriptor to a socket with an address in the privileged port space. The `ruserok()` function is used by servers to authenticate clients requesting service with `rcmd()`. All three functions are present in the same file and are used by the `rshd` server (see the Utilities Reference), among others.

The `rresvport()` function obtains a socket with a privileged address bound to it. This socket can be used by `rcmd()` and several other functions.

Returns:

A valid, bound socket descriptor, or -1 if an error occurs (`errno` is set).

Errors:

The error code EAGAIN is overloaded to mean “All network ports in use.”
**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*rcmd(), ruserok()*

*rshd* in the *Utilities Reference*
Rselect()

Check for descriptors that are ready for reading or writing (via a SOCKS server)

Synopsis:

```c
int Rselect( int width,
             fd_set * readfds,
             fd_set * writefds,
             fd_set * exceptionfds,
             struct timeval * timeout);
```

Arguments:

- **width**: The number of descriptors to check in the given sets. Only the descriptors from 0 through \((\text{width}-1)\) in the descriptor sets are examined. Therefore, the value of `width` must be at least as large as:

\[
\text{(highest valued file descriptor in the sets) + 1}
\]

- **readfds**: NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for reading. The function replaces the set with the file descriptors that are actually ready for reading.

- **writefds**: NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for writing. The function replaces the set with the file descriptors that are actually ready for writing.

- **exceptionfds**: NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that have an exceptional condition pending. The function replaces the set with the file descriptors that actually have an exceptional condition pending.

- **timeout**: NULL, or a pointer to a `timeval` that specifies how long to wait for the selection to complete.

Library:

- **libsocks**

Use the `-l socks` option to `qcc` to link against this library.
**Rselect()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The `Rselect()` function is a cover function for `select()` — the difference is that `Rselect()` does its job via a SOCKS server. For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.

**Returns:**

The number of ready descriptors in the descriptor sets, 0 if the `timeout` expired, or -1 if an error occurs (`errno` is set).

**Classification:**

SOCKS

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`select()`

SOCKS — A Basic Firewall
**Synopsis:**

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_attach( rsrc_request_t * list,
                     int count );
```

**Arguments:**

- `list` An array of `rsrc_request_t` structures that describe the resources that you want to reserve; see below.
- `count` The number of entries in the array.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The resource database manager allocates and keeps track of system resources i.e. it manages these resources. The system resources currently tracked are:

- memory
- IRQs
- DMA channels
- I/O ports.

Major and minor device numbers are handled with separate `rsrdbmgr_devno_attach()` and `rsrdbmgr_devno_detach()` functions.

There are two main functions that drivers can use to communicate with the resource database:
rsrcremgr_attach()

- rsrcremgr_attach()
- rsrcremgr_detach()

The rsrcremgr_attach() function reserves a resource range(s) from the database of available resources for a process. Other processes can’t reserve this resource range until the resource is returned to the system (usually with the rsrcremgr_detach() call). The requested resources are returned in a list of _rsrc_request structures with the start and end fields filled in. The number of resources requested is specified in count.

Reserving the resources doesn’t give you access to them; you still have to use mmap(), InterruptAttach(), or another means.

When you’re finished with the resource, you must return it to the system. The easiest way to return the resource is to call rsrcremgr_detach() with the same start, end, and type (via the flags field) that were issued for the resource.

rsrc_request_t structure

The resource requests structure looks like this:

```c
typedef struct _rsrc_request {
    uint64_t length;
    uint64_t align;
    uint64_t start;
    uint64_t end;
    uint32_t flags;
    uint32_t zero[3]; /* Reserved */
} rsrc_request_t;
```

The members include:

- **length**: The length of the resource that you want to reserve. You must set this member.
- **align**: The alignment of the resource.
- **start, end**: The range of resource that you want to reserve.
flags

The type of the resource, as well as flags that affect the request. You must set this member to be one of the following resource types (defined in `<sys/rsrcdbmgr.h>`):

- RSRCDBMGR_DMA_CHANNEL — DMA channel
- RSRCDBMGR_IO_PORT — I/O port address
- RSRCDBMGR_IRQ — Interrupt address
- RSRCDBMGR_MEMORY — Memory address
- RSRCDBMGR_PCI_MEMORY — PCI memory address

You can OR in the following flags (also defined in `<sys/rsrcdbmgr.h>`):

- RSRCDBMGR_FLAG_ALIGN — the contents of the `align` field are valid, and the requested resource starts with the given alignment.
- RSRCDBMGR_FLAG_RANGE — the contents of the `start` and `end` fields are valid, and the requested resource is in the range `start` to `end`, inclusive.
- RSRCDBMGR_FLAG_SHARE — other processes can have access to an allocated resource.
- RSRCDBMGR_FLAG_TOPDOWN — start the search for a free resource block from `end`. If you also set RSRCDBMGR_FLAG_RANGE, this flag makes the search start from the `end` of the available range.

Returns:

EOK, or -1 if an error occurred (`errno` is set).

Errors:

- EAGAIN The resource request can’t be filled.
- EINVAL Invalid argument.
- ENOMEM Insufficient memory to allocate internal data structures.
Examples:

When you start the system, the startup code and special programs that know how to probe the hardware call `rsrcdbmgr_create()` to register the hardware in the resource database. The following examples don’t do this seeding, so they’ll fail with an error code of EINVAL.

Example 1

```c
/*
 * Request one DMA Channel, with length 1, from the
 * entire range of available DMA channel resources.
 */
#include <stdio.h>
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

int main(int argc, char **argv) {
    int count;
    rsrc_request_t req;

    memset(&req, 0, sizeof(req));
    req.length = 1;
    req.flags = RSRCDBMGR_DMA_CHANNEL;
    count = 1;

    if (rsrcdbmgr_attach( &req, count) == -1) {
        perror("Problem attaching to resource ");
        exit(1);
    }

    printf("You can use DMA channel 0x%llx \n",
           req.start);

    ...
    /* Do something with the acquired resource */
    ...

    /* To return the resource to the database: */
    if (rsrcdbmgr_detach( &req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }

    return(0);
}
```
Example 2

```c
/*
 * Request memory that's 4-byte aligned
 * and has a length of 50.
 */

#include <stdio.h>
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

int main(int argc, char **argv) {
    int count;
    rsrc_request_t req;

    memset(&req, 0, sizeof(req));
    req.align = 4;
    req.length = 50;
    req.flags = RSRCDBMGR_FLAG_ALIGN | RSRCDBMGR_MEMORY;
    count = 1;

    if (rsrcdbmgr_attach(&req, count) == -1) {
        perror("Problem attaching to resource ");
        exit(1);
    }

    printf("You can use memory from 0x%llx 0x%llx inclusive. \n",
            req.start, req.end);

    ...  /* Do something with the acquired resource */
    ...

    /* To return the resource to the database: */
    if (rsrcdbmgr_detach(&req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }

    return(0);
}
```

Example 3

```c
/*
 * Request two resources:
 * I/O port 0 and an IRQ in the range 10-12
 * from the available resources.
 */
#include <stdio.h>
#include <sys/rsrcdbmgr.h>
```
#include <sys/rsrcecbmgr.h>

int main(int argc, char **argv) {
    int count;
    rsrc_request_t req[2];

    memset(req, 0, 2*sizeof(*req));
    req[0].start = 0;
    req[0].end = 0;
    req[0].length = 1;
    req[0].flags = RSRCDBMGR_FLAG_RANGE | RSRCDBMGR_IO_PORT;

    req[1].start = 10;
    req[1].end = 12;
    req[1].length = 1;
    req[1].flags = RSRCDBMGR_FLAG_RANGE | RSRCDBMGR_IRQ;
    count = 2;

    if (rsrcdbmgr_attach(req, count) == -1) {
        perror("Problem attaching to resource ");
        exit(1);
    }

    printf("You can use io-port 0x%llx \n",
        req[0].start);
    printf("You can use irq 0x%llx \n",
        req[1].start);

    ... /* Do something with the acquired resource */
    ...

    /* To return the resource to the database: */
    if (rsrcdbmgr_detach(req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }

    return(0);
}

Classification:

QNX Neutrino
rsrcrealmgr_attach()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

rsrcrealmgr_create(), rsrcdbmgr_detach(), rsrcdbmgr_destory()
rsrcrealmgr_devno_attach(), rsrcdbmgr_devno_detach()
rsrcrealmgr_query()
rsrclbmgr_create()  © 2007, QNX Software Systems GmbH & Co. KG.

Create a system resource

Synopsis:

#include <sys/rsrclbmgr.h>
#include <sys/rsrclbmsg.h>

int rsrcdbmgr_create( rsrc_alloc_t *item,
                       int count );

Arguments:

item  An array of rsrc_alloc_t structures that describe the resources that you want to create; see below.

count  The number of entries in the array.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The rsrcdbmgr_create() function creates one or more system resources. If the function completes successfully, count resources are returned in item.

rsrc_alloc_t structure

The structure of a basic resource request looks like this:

typedef struct _rsrc_alloc {
    uint64_t start; /* Start of resource range */
    uint64_t end;  /* End of resource range */
    uint32_t flags; /* Resource type | Resource flags */
} rsrc_alloc_t;

The members include:

start, end  The resource range.
flags

The type of the resource, as well as flags that affect the request. You must set this member to be one of the following resource types (defined in `<sys/rsrcreg.h>`):

- RSRCDBMGR_DMA_CHANNEL — DMA channel
- RSRCDBMGR_IO_PORT — I/O port address
- RSRCDBMGR_IRQ or RSRCMGR_IRQ — interrupt address
- RSRCDBMGR_MEMORY — Memory address
- RSRCDBMGR_PCI_MEMORY — PCI memory address

You can OR in the following bits (also defined in `<sys/rsrcreg.h>`):

- RSRCDBMGR_FLAG_NOREMOVE — don’t remove this resource when the process dies.
- RSRCDBMGR_FLAG_RSVP — create and reserve a resource with a higher priority than other resources. The resource is given out only when there are no other valid ranges available.

You must set all the members.

Returns:

EOK, or -1 if an error occurred (errno is set).

Errors:

EAGAIN The resource request can’t be created.
EINVAL Invalid argument.
ENOMEM Insufficient memory to allocate internal data structures.
Examples:

```c
/*
 * Create two resources:
 * 0-4K memory allocation and 5 DMA channels.
 */
#include <stdio.h>
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

int main(int argc, char **argv) {
    rsrc_alloc_t alloc[2];
    memset(alloc, 0, 2* sizeof(*alloc));
    alloc[0].start = 0;
    alloc[0].end = 4*1024;
    alloc[0].flags = RSRCDBMGR_MEMORY;
    alloc[1].start = 1;
    alloc[1].end = 5;
    alloc[1].flags = RSRCDBMGR_DMA_CHANNEL;

    /* Allocate resources to the system. */
    if (rsrcdbmgr_create( alloc, 2 ) == -1) {
        perror("Problem creating resources 
");
        exit(1);
    }

    /* Do something with the created resource */
    ...

    /* Remove the allocated resources. */
    rsrcdbmgr_destroy ( alloc, 2 );
    return(0);
}
```

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
### `rsrcremgr_create()`

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`rsrcremgr_attach()`, `rsrcremgr_destroy()`
rsrcregmgr_destroy()  © 2007, QNX Software Systems GmbH & Co. KG.

Destroy a system resource

Synopsis:

#include <sys/rsrcregmgr.h>
#include <sys/rsrcregmsg.h>

int rsrcregmgr_destroy( rsrc_alloc_t *item,
                         int count);

Arguments:

item  An array of rsrc_alloc_t structures that describe the resources that you want to destroy. For more information about this structure, see the documentation for rsrcregmgr_create().

count  The number of entries in the array.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The rsrcregmgr_destroy() function removes count system resources that are defined in the array item.

Returns:

EOK  Success.

-1  An error occurred; errno is set.

Errors:

EINV  Invalid argument, or the resource is in use.

ENOMEM  Insufficient memory to allocate internal data structures.
Examples:

See the example in `rsrdbmgr_create()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`rsrdbmgr_attach()`, `rsrdbmgr_create()`, `rsrdbmgr_detach()`
rsrcreqmgr_detach()

Synopsis:
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

int rsrcreqmgr_detach( rsrc_request_t *list,
int count );

Arguments:

list An array of rsrc_request_t structures that describe the resources that you want to return. For information about this structure, see the documentation for rsrcreqmgr_attach().

count The number of entries in the array.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The rsrcreqmgr_detach() function returns count resources in list to the database of available system resources. You must return the resource with the same start, end, and flags (type) that were issued for the resource when it was allocated with rsrcreqmgr_attach().

Returns:

EOK Success.

-1 An error occurred; errno is set.
Errors:

- **EINVAL**: Invalid argument, or the resource is in use by a process, isn’t found in the database, or can’t be returned to the system.
- **ENOMEM**: Insufficient memory to allocate internal data structures.

Examples:

See the examples in `rsrcdbmgr_attach()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`rsrcdbmgr_attach()`, `rsrcdbmgr_destroy()`
Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

dev_t rsrdbmgr_devno_attach( const char * name,
    int minor_request,
    int flags );
```

Arguments:

- **name**: The name of the class of devices that you want to get the major number for. This string can be anything, but various names are defined in in `<sys/ftype.h>`; see “Class names,” below.

- **minor_request**: The minor device number that you want to reserve, or -1 to let the system assign the next available minor number.

- **flags**: Presently, there are no flags; pass zero for this argument.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `rsrdbmgr_devno_attach()` reserves a device number that consists of:

- a major number that corresponds to the given device class. If there isn’t already a major number associated with the class, a new major number is assigned to it.

- a minor number that’s based on `minor_request`. If `minor_request` is -1, the function returns the first free minor number in the specified class.
There’s a maximum of 64 major numbers (0 through 63) on the system, and a maximum of 1024 minor numbers (0 through 1023) per major number.

Major and minor numbers are used only by resource managers and are exposed through the rdev member of the iofunc_attr_t structure, and correspondingly the st_rdev member of the stat structure. They aren’t required for proper operation; on simple devices, an entry will be simulated for you.

**Class names**

As mentioned about, the name of the class of devices can be anything. The following class names are defined in `<sys/ftype.h>`:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MAJOR_PATHMGR</td>
<td>&quot;pathmgr&quot;</td>
<td>Used only by the path manager</td>
</tr>
<tr>
<td>_MAJOR_DEV</td>
<td>&quot;dev&quot;</td>
<td>Devices in /dev with only one instance (e.g. /dev/tty)</td>
</tr>
<tr>
<td>_MAJOR_BLK_PREFIX</td>
<td>&quot;blk-&quot;</td>
<td>All block devices (e.g. /dev/hd[0-9]*) would be &quot;blk-hd&quot;)</td>
</tr>
<tr>
<td>_MAJOR_CHAR_PREFIX</td>
<td>&quot;char-&quot;</td>
<td>All character devices (e.g. /dev/ser[0-9]*) would be &quot;char-ser&quot;)</td>
</tr>
<tr>
<td>_MAJOR_FSYS</td>
<td>&quot;fsys&quot;</td>
<td>All filesystems</td>
</tr>
</tbody>
</table>

**Returns:**

A dev_t object that contains the major and minor numbers, or -1 if an error occurs (errno is set).
You can extract the major and minor number values from the dev_t object by using the major() and minor() macros defined in "<sys/types.h>". For more information, see the documentation for stat().

Errors:

EINVAL   Invalid argument.

Examples:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

char *dev_name;
int myminor_request, flags=0;
dev_t major_minor;

major_minor = rsrcdbmgr_devno_attach
       ( dev_name, myminor_request, flags );

rsrcdbmgr_devno_detach( major_minor, flags );
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

If your application calls this function, it must run as root.

See also:

iofunc_attr_t, rsrdbmgr_attach(), rsrdbmgr_devno_detach(), stat()
rsrctdbmgr_devno_detach() © 2007, QNX Software Systems GmbH & Co. KG.

Detach a major and minor number

Synopsis:

```
#include <sys/rsrctdbmgr.h>
#include <sys/rsrctdbmsg.h>

int rsrctdbmgr_devno_detach( dev_t devno,
                                int flags );
```

Arguments:

devo A dev_t object that was returned by
rsrctdbmgr_devno_attach().

flags Presently, there are no flags; pass zero for this argument.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The function rsrctdbmgr_devno_detach() detaches device number that
was attached with rsrctdbmgr_devno_attach().

Returns:

EOK Success.

-1 An error occurred.

Examples:

```
#include <sys/rsrctdbmgr.h>
#include <sys/rsrctdbmsg.h>

dev_t dev_num;
int flags=0;

:

rsrctdbmgr_devno_detach( dev_num, flags );
```
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

rsrcdbmgr_attach(), rsrcdbmgr_devno_attach()
rsrdbmgr_query()

Query the resource database

Synopsis:

```
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_query( rsrc_alloc_t *list,
    int listcnt,
    int start,
    uint32_t type);
```

Arguments:

- **list**: NULL, or an array of `rsrc_alloc_t` structures that the function can fill with information about the resources that it finds. For more information about this structure, see the documentation for `rsrdbmgr_create()`.
- **listcnt**: The number of entries in the array.
- **start**: The index that you want to start searching at.
- **type**: The type of resource that you want to query; one of the following (defined in `<sys/rsrdbmgr.h>`):
  - `RSRCDMBGR_DMA_CHANNEL` — DMA channel
  - `RSRCDMBGR_IO_PORT` — I/O port address
  - `RSRCDMBGR_IRQ` or `RSRCDMBGR_IRQ` — interrupt address
  - `RSRCDMBGR_MEMORY` — Memory address
  - `RSRCDMBGR_PCI_MEMORY` — PCI memory address

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:
The `rsrcdbmgr_query()` function queries the database for `listcnt` count of `type` resources in use, beginning at the index `start`. If you make the query with a non-NULL `list`, then the function stores a maximum of found `listcnt` resources in the array.

Returns:
If `list` is NULL or `listcnt` is 0, then the function returns the number of resources of `type` in the database.

If `list` is non-NULL, then the function returns the number of `type` resources available in the system.

If an error occurs, the function returns -1 and sets `errno`.

Errors:
EINV AL Valid argument
ENOMEM Insufficient memory to allocate internal data structures.

Examples:
List all of the memory resource blocks available in the system:

```c
rsrc_alloc_t  list[20];
int  size, count = 0, start = 0;

while (1) {
    count = rsrcdbmgr_query( list, 20, start, RSRCDBMGR_MEMORY );
    if (count == -1)
        break;
    size = min( count-start, 20 ); /* In case more than 20 blocks 
        were returned. */
    printf( "Retrieved %d of a possible %d resource blocks", \
            size, count);
    for (count=0; count<size; count++) {
        printf( "RSRC[%d] Start %d End %d \n", \
                start+count, list[count].start, list[count].end);
    }
    start += size; /* Loop again, in case there are more than
                    */
```
rsrCdBMgr_query()

20 blocks. */

}

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

rsrCdBMgr_attach()
Synopsis:

```c
#include <unistd.h>

int ruserok( char * rhost,
             int superuser,
             char * ruser,
             char * luser );
```

Arguments:

- `rhost`: The name of the remote host, as returned by `gethostbyaddr()`.
- `superuser`: Nonzero if the local user is the superuser, zero otherwise.
- `ruser`: The name of the remote user.
- `luser`: The name of the local user.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `ruserok()` routine checks the identity of a remote host. It’s used by servers to authenticate clients requesting service with `rcmd()`.

The `rcmd()`, `rresvport()`, and `ruserok()` functions are used by the `rshd` server (see the Utilities Reference), among others.

The `ruserok()` function takes a remote host’s name (as returned by the `gethostbyaddr()` routine), two user names, and a flag indicating whether the local user’s name is that of the superuser. Then, if the user is not the superuser, it checks the file `/etc/hosts.equiv` (described in the Utilities Reference).

If that lookup isn’t done, or is unsuccessful, the `.rhosts` file in the local user’s home directory is checked to see if the request for service
is allowed. If this file is owned by anyone other than the user or the superuser, or if it’s writable by anyone other than the owner, the check automatically fails.

If the local domain obtained from gethostname() is the same as the remote domain, only the machine name need be specified.

Returns:

0 The machine name is listed in the hosts.equiv file, or the host and remote username were found in the .rhosts file.

-1 An error occurred (errno is set).

Errors:

The error code EAGAIN is overloaded to mean “All network ports in use.”

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

gethostbyaddr(), gethostname(), rcmd(), resvport()

/etc/hosts.equiv, rshd in the Utilities Reference
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td>abort() to expm1f()</td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td>fabs() to hypotf()</td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to ltrunc()</td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td>main() to outle32()</td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td>pathconf() to ruserok()</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td>sbrk() to system()</td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td>tan() to ynf()</td>
</tr>
</tbody>
</table>
**sbrk()**

Set the allocation break value

**Synopsis:**

```c
#include <unistd.h>

void* sbrk( int increment );
```

**Arguments:**

- `increment` The amount by which to increase the current break value. This increment may be positive or negative.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `break` value is the address of the first byte of unallocated memory. When a program starts execution, the break value is placed following the code and constant data for the program. As memory is allocated, this pointer advances when there is no free block large enough to satisfy an allocation request. The `sbrk()` function sets a new break value for the program by adding the value of `increment` to the current break value. Newly allocated space is set to zero.

The variable `_ambilksiz` (defined in `<stdlib.h>`) contains the default increment. This value may be changed by a program at any time.

**Returns:**

A pointer to the start of the new block of memory for success, or `-1` if an error occurs (`errno` is set).
Errors:

EAGAIN  The total amount of system memory available for allocation to this process is temporarily insufficient. This may occur although the space requested is less than the maximum data segment size.

ENOMEM  The requested change allocated more space than allowed, is impossible since there’s insufficient swap space available, or it caused a memory allocation conflict.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

#define alloc( x, y ) y = sbrk( x );

int main( void )
{
    void* brk;
    brk = sbrk( 0x3100 );
    printf( "New break value after sbrk( 0x3100 ) \t%p\n", brk );
    brk = sbrk( 0x0200 );
    printf( "New break value after sbrk( 0x0200 ) \t%p\n", brk );
    return EXIT_SUCCESS;
}
```

Classification:

Legacy Unix

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued…
Caveats:

The behavior of `sbrk()` is unspecified if an application also uses any other memory functions (such as `malloc()`, `mmap()`, and `free()`). The `sbrk()` function has been used in specialized cases where no other memory allocation function provided the same capability. Use `mmap()` instead because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

The value of the argument to `sbrk()` is rounded up for alignment with eight-byte boundaries.

Setting the break may fail due to a temporary lack of swap space. It isn’t possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting `getrlimit()`.

See also:

`_amblksz`, `brk()`, `_btext`, `calloc()`, `_edata`, `_end`, `errno`, `_etext`, `free()`, `malloc()`, `realloc()`
scalb(), scalbf()

Load the exponent of a radix-independent floating point number

Synopsis:

```c
#include <math.h>

double scalb( double x,  
             double n );

float scalbf( float x,  
              float n );
```

Arguments:

- `x` The floating point number that you want to multiply by the exponent.
- `n` The exponent to apply to the radix of the machine’s floating-point arithmetic.

Library:

libm

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute \( x \times r^n \), where \( r \) is the radix of the machine’s floating point arithmetic and \( n \) is a finite number. When \( r \) is 2, `scalb()` is equivalent to `ldexp()`.

We recommend that you use `scalbn()` because it computes by manipulating exponents, instead of using mock multiplications or additions. The `scalbf()` function is based on an ANSI draft; for more portable code, use `scalbnf()` instead.

Returns:

\( x \times r^n \)
If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set \texttt{errno} to 0, call the function, and then check \texttt{errno} again. These functions don’t change \texttt{errno} if no errors occurred.

**Examples:**

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;
    a = 10;
    b = 2;
    c = scalb(a, b);
    d = sqrt(c/a);
    printf("Radix of machines fp arithmetic is %f \n", d);
    printf("So %f = %f * (%f ^ %f) \n", c, a, d, b);
    return(0);
}
```

produces the output:

```
Radix of machines fp arithmetic is 2.000000
So 40.000000 = 10.000000 * (2.000000 ^ 2.000000)
```

**Classification:**

\texttt{scalb()} is POSIX 1003.1 XSI; \texttt{scalbf()} is ANSI (draft)

**Safety**

- Cancellation point: No
- Interrupt handler: No

*continued...*
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

*ldexp(*), *scalbn(*)
scalbn(), scalbnf()  
Load the exponent of a radix-independent floating point number

**Synopsis:**
```c
#include <math.h>

double scalbn ( double x, 
           int n );

float scalbnf ( float x, 
           int n );
```

**Arguments:**
- `x` The floating point number that you want to multiply by the exponent.
- `n` The exponent to apply to the radix of the machine’s floating-point arithmetic.

**Library:**
```
libm
```
Use the `-l m` option to `qcc` to link against this library.

**Description:**
The `scalbn()` and `scalbnf()` functions compute \( x \times r^n \), where \( r \) is the radix of the machine’s floating point arithmetic.

**Returns:**
\( x \times r^n \)

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
Examples:

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;
    a = 10;
    b = 2;
    c = scalbn(a, b);
    d = sqrt(c/a);
    printf("Radix of machines fp arithmetic is %f \n", d);
    printf("So %f = %f * (%f ^ %f) \n", c, a, d, b);
    return(0);
}
```

produces the output:

```
Radix of machines fp arithmetic is 2.000000
So 40.000000 = 10.000000 * (2.000000 ^ 2.000000)
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
scalbn(), scalbnf()

See also:

scalb()
**Synopsis:**

```
#include <malloc.h>

void* _scalloc( size_t size );
```

**Arguments:**

- `size` The number of bytes to allocate.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `_scalloc()` functions allocate space for an array of length `size` bytes. Each element is initialized to 0.

You must use `_sfree()` to deallocate the memory allocated by `_scalloc()`.

**Returns:**

A pointer to the start of the allocated memory, or NULL if there’s insufficient memory available or if the `size` is zero.

**Classification:**

QNX Neutrino
**_scalloc()_**

_C Library — S September 10, 2007_

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`calloc(), free(), realloc(), _sfree(), _smalloc(), _srealloc()`
Synopsis:

```c
#include <sys/types.h>
#include <sys/dir.h>

int scandir( char * dirname,
             struct direct * (* namelist[]),
             int (*select) (struct dirent *),
             int (*compar)(const void *,const void *) );
```

Arguments:

- `dirname` The name of the directory that you want to scan.
- `namelist` A pointer to a location where `scandir()` can store a pointer to the array of directory entries that it builds.
- `select` A pointer to a user-supplied subroutine that `scandir()` calls to select which entries to included in the array. The `select` routine is passed a pointer to a directory entry and should return a nonzero value if the directory entry is to be included in the array.

If `select` is `NULL`, all the directory entries are included.

- `compar` A pointer to a user-supplied subroutine that’s passed to `qsort()` to sort the completed array. If this pointer is `NULL`, the array isn’t sorted.

You can use `alphasort()` as the `compar` parameter to sort the array alphabetically.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `scandir()` function reads the directory `dirname` and builds an array of pointers to directory entries, using `malloc()` to allocate the space. The `scandir()` function returns the number of entries in the array, and stores a pointer to the array in the location referenced by `namelist`.

You can deallocate the memory allocated for the array by calling `free()`. Free each pointer in the array, and then free the array itself.

**Returns:**

The number of entries in the array, or -1 if the directory can’t be opened for reading, or `malloc()` can’t allocate enough memory to hold all the data structures.

**Classification:**

Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

alphasort(), closedir(), free(), malloc(), opendir(), qsort(), readdir(),
rewinddir(), seekdir(), telldir()
**Synopsis:**

```c
#include <stdio.h>

int scanf( const char* format,
            ... );
```

**Arguments:**

- `format` A string that controls the format of the input, as described below. The formatting string determines what additional arguments you need to provide.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `scanf()` function scans input from `stdin` under control of the `format` argument, assigning values to the remaining arguments.

**Format control string**

The format control string consists of zero or more `format directives` that specify what you consider to be acceptable input data. Subsequent arguments are pointers to various types of objects that the function assigns values to as it processes the format string.

A format directive can be a sequence of one or more whitespace characters or:

- **multibyte characters**
  
  Any character in the `format` string, other than a whitespace character or the percent character (%), that isn’t part of a conversion specifier.
conversion specifiers

A sequence of characters in the format string that begins with a percent character (%) and is followed by:

- an optional assignment suppression indicator: the asterisk character (*)
- an optional decimal integer that specifies the maximum field width to be scanned for the conversion
- an optional type length specification; one of h, l, or l
- a character that specifies the type of conversion to be performed; one of the characters: c, d, e, f, g, i, n, o, p, s, u, x, x, [...]

As each format directive in the format string is processed, the directive may successfully complete, fail because of a lack of input data, or fail because of a matching error as defined by the directive.

If end-of-file is encountered on the input data before any characters that match the current directive have been processed (other than leading whitespace, where permitted), the directive fails for lack of data.

If end-of-file occurs after a matching character has been processed, the directive is completed (unless a matching error occurs), and the function returns without processing the next directive.

If a directive fails because of an input character mismatch, the character is left unread in the input stream.

Trailing whitespace characters, including newline characters, aren’t read unless matched by a directive. When a format directive fails, or the end of the format string is encountered, the scanning is completed, and the function returns.

When one or more whitespace characters (space, horizontal tab \t, vertical tab \v, form feed \f, carriage return \r, newline or linefeed \n) occur in the format string, input data up to the first non-whitespace character is read, or until no more data remains. If no whitespace characters are found in the input data, the scanning is complete, and the function returns.
An ordinary character in the format string is expected to match the same character in the input stream.

**Conversion specifiers**

A conversion specifier in the format string is processed as follows:

- For conversion types other than `i`, `c` and `n`, leading whitespace characters are skipped.

- For conversion types other than `n`, all input characters, up to any specified maximum field length, that can be matched by the conversion type are read and converted to the appropriate type of value; the character immediately following the last character to be matched is left unread; if no characters are matched, the format directive fails.

- Unless you specify the assignment suppression indicator (`*`), the result of the conversion is assigned to the object pointed to by the next unused argument (if assignment suppression was specified, no argument is skipped); the arguments must correspond in number, type and order to the conversion specifiers in the format string.

**Type length specifiers**

A type length specifier affects the conversion as follows:

- `hh` causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` format conversion to assign the converted value to an object of type `signed char` or `unsigned char`.

- `h` causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` (integer) format conversion to assign the converted value to an object of type `short` or `unsigned short`.

- `j` causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` conversion to assign the converted value to an object of type `intmax_t` or `uintmax_t`.

- `l` (“el”) causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` (integer) conversion to assign the converted value to an object of type `long` or `unsigned long`.
- `l` ("el") causes an `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` conversion to assign the converted value to an object of type `double`.

- `l` ("el") causes a `c`, `s` or `I` conversion to assign the converted value to an object of type `wchar_t`.

- `ll` (double “el”) causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` format conversion to assign the converted value to an object of type `long long` or `unsigned long long`.

- `L` causes an `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` conversion to assign the converted value to an object of type `long double`.

- `t` causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` conversion to assign the converted value to an object of type `ptrdiff_t` or to the corresponding unsigned type.

- `z` causes a `d`, `i`, `o`, `u`, `x`, `X` or `n` conversion to assign the converted value to an object of type `size_t` or to the corresponding signed integer type.

### Conversion type specifiers

The valid conversion type specifiers are:

- `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G`
  A floating-point number, infinity, or NaN, all of which have a format as expected by `strtod()`.
  The argument is assumed to point to an object of type `float`.

- `c`
  Any sequence of characters in the input stream of the length specified by the field width, or a single character if you don’t specify a field width.
  The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence, without a terminating NUL character (`\0`). For a single character assignment, a pointer to a single object of type `char` is sufficient.

When an `l` ("el") qualifier is present, a sequence of characters are converted from the initial shift state to
wchar_t wide characters as if by a call to mbtowc(). The conversion state is described by a mbstate_t object.

d A decimal integer with a format as expected by strtol() and a base of 10. The argument is assumed to point to an object of type int.

i An optionally signed integer with a format as expected by strtol() and a base of 0. The argument is assumed to point to an object of type int.

n No input data is processed. Instead, the number of characters that have already been read is assigned to the object of type int that’s pointed to by the argument. The number of items that have been scanned and assigned (the return value) isn’t affected by the n conversion type specifier.

o An optionally signed octal integer with a format as expected by strtoul() and a base of 8. The argument is assumed to point to an object of type int.

p A hexadecimal integer, as described for x conversions below. The converted value is taken as a void * and then assigned to the object pointed to by the argument.

s A sequence of non-whitespace characters. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence of char, signed char or unsigned char and a terminating NUL character, which the conversion operation adds.

When an l ("el") qualifier is present, a sequence of characters are converted from the initial shift state to wchar_t wide characters as if by a call to mbtowc(). The conversion state is described by a mbstate_t object.

u An unsigned decimal integer, consisting of one or more decimal digits. The argument is assumed to point to an object of type unsigned int.
scanf()  

x, X  A hexadecimal integer, with a format as expected by `strtoul()` when `base` is 16. The argument is assumed to point to an object of type `unsigned`.

[  Matches the `scanset`, a nonempty sequence of characters. The argument is assumed to point to the first element of a character array of sufficient size to contain the sequence and a terminating NUL character, which the conversion operation adds.

When an `l` ("el") qualifier is present, a sequence of characters are converted from the initial shift state to `wchar_t` wide characters as if by a call to `mbrtowc()` with `mbstate` set to 0. The argument is assumed to point to the first element of a `wchar_t` array of sufficient size to contain the sequence and a terminating NUL character, which the conversion operation adds.

The conversion specification includes all characters in the scanlist between the beginning [ and the terminating ]. If the conversion specification starts with `[^]`, the scanlist matches all the characters that aren’t in the scanlist. If the conversion specification starts with `[` or `[^]`, the `]` is included in the scanlist. (To scan for ] only, specify `%[ ]`.)

%  A % character (The entire specification is %).

A conversion type specifier of % is treated as a single ordinary character that matches a single % character in the input data. A conversion type specifier other than those listed above causes scanning to terminate, and the function to returns with an error.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning stopped by reaching the end of the input stream before storing any values.
Examples:

The line:

```c
scanf( "%s%f%3hx%d", name, &hexnum, &decnum )
```

with input:

```
some_string 34.555e-3 abc1234
```

copies "some_string" into the array name, skips 34.555e-3, assigns 0xabc to hexnum and 1234 to decnum. The return value is 3.

The program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char string1[80], string2[80];
    memset( string1, 0, 80 );
    memset( string2, 0, 80 );
    scanf( "%[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]="%\n", string1, string2 );
    printf( "%s\n", string1 );
    printf( "%s\n", string2 );
    return EXIT_SUCCESS;
}
```

with input:

They may look alike, but they don’t perform alike.

assigns "They may look alike" to string1, skips the comma (the "%2s" matches only the comma; the following blank terminates that field), and assigns " but they don’t perform alike." to string2.

To scan a date in the form “Friday March 26 1999”:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    int day, year;
    char weekday[10], month[12];
    int retval;

    memset( weekday, 0, 10 );
    memset( month, 0, 12 );

    retval = scanf( "%s %s %d %d",
                    weekday, month, &day, &year );
    if( retval != 4 ) { 
        printf( "Error reading date.\n" );
        printf( "Format is: Friday March 26 1999\n" );

        return EXIT_FAILURE;
    }

    printf( "weekday: %s\n", weekday );
    printf( "month: %s\n", month );
    printf( "day: %d\n", day );
    printf( "year: %d\n", year );

    return EXIT_SUCCESS;
}

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

fscanf(), fwscanf(), sscanf(), swscanf(), vfscanf(), vwscanf(),
vscanf(), vsscanf(), vswscanf(), vfwscanf(), vsscanf()
**sched_getparam()**

Get the current priority of a process

**Synopsis:**

```c
#include <sched.h>

int sched_getparam( pid_t pid,
                     struct sched_param *param );
```

**Arguments:**

- **pid**: The ID of the process whose priority you want to get, or 0 to get it for the current process.
- **param**: A pointer to a `struct sched_param` that the function fills with the scheduling parameters.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sched_getparam()` function gets the current priority of the process specified by `pid`, and puts it in the `sched_priority` member of the `struct sched_param` structure pointed to by `param`.

If `pid` is zero, the priority of the calling process is returned.

**Returns:**

- **0**: Success
- **-1**: An error occurred (`errno` is set).

**Errors:**

- **EPERM**: The calling process doesn’t have sufficient privilege to get the priority.
- **ESRCH**: The process `pid` doesn’t exist.
Examples:

```c
#include <sched.h>
#include <stdio.h>

#define PRIORITY_ADJUSTMENT 5

int main (void)
{
    int max_priority;
    struct sched_param param;

    /* Find out the MAX priority for the FIFO Scheduler */
    max_priority = sched_get_priority_max(SCHED_FIFO);

    /* Find out what the current priority is. */
    sched_getparam(0, &param);

    printf("The assigned priority is \%d.\n", param.sched_priority);
    printf("The current priority is \%d.\n", param.sched_curpriority);

    param.sched_priority = ((param.sched_curpriority +
                                PRIORITY_ADJUSTMENT) <= max_priority) ?
                        (param.sched_curpriority + PRIORITY_ADJUSTMENT) : -1;

    if (param.sched_priority == -1)
    {
        printf("Cannot increase the priority by \%d. Try a smaller value\n",
                PRIORITY_ADJUSTMENT);
        return(0);
    }

    sched_setscheduler (0, SCHED_FIFO, &param);

    sched_getparam(0, &qparam);
    printf("The newly assigned priority is \%d.\n",
            param.sched_priority);
    printf("The current priority is \%d.\n",
            param.sched_curpriority);
    return(0);
}
```

Classification:

POSIX 1003.1 PS
sched_getparam()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

Currently, the implementation of `sched_getparam()` isn’t 100% POSIX 1003.1-1996. The `sched_getparam()` function returns the scheduling parameters for thread 1 in the process `pid`, or for the calling thread if `pid` is 0.

If you depend on this in new code, *it will not be portable*. POSIX 1003.1 says `sched_getparam()` should return -1 and set `errno` to EPERM in a multithreaded application.

**See also:**

`errno`, `getprio()`, `sched_get_priority_max()`,
`sched_get_priority_min()`, `sched_getscheduler()`, `sched_param`,
`sched_setscheduler()`, `sched_setparam()`, `sched_setscheduler()`, `sched_yield()`, `setprio()`
calculate the allowable priority for the scheduling policy

Synopsis:

```c
#include <sched.h>

int sched_get_priority_adjust( int prio,
                              int policy,
                              int adjust );
```

Arguments:

- `prio` The original priority value. If negative, the priority of the calling thread is used.
- `policy` The scheduling algorithm being used. The valid arguments are listed in `sched_get_priority_max()`. If `policy` is `SCHED_NOCHANGE`, the function uses the algorithm of the calling thread.
- `adjust` The priority change, relative to `prio`. A value of +10 results in a final priority of `prio`+10, provided that this amount of adjustment is allowed.

Library:

- `libc`

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `sched_get_priority_adjust()` function calculates the requested priority change relative to another thread and returns the allowable value.

This function makes it easier for you to set relative priorities in order to ensure proper precedence.
sched_get_priority_adjust()

Returns:

>0 The allowed priority value. The value will never exceed the range of values allowed by sched_get_priority_min() and sched_get_priority_max().

<0 Failure; the negative of the errno value.

Errors:

EINVAL The value of the policy parameter doesn’t represent a defined scheduling policy.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

errno, sched_getparam(), sched_get_priority_max(), sched_get_priority_min(), sched_setparam(), sched_getscheduler(), sched_setscheduler()
Get the maximum priority for the scheduling policy

**Synopsis:**

```c
#include <sched.h>

int sched_get_priority_max( int policy );
```

**Arguments:**

- `policy` The scheduling policy, which must be one of:
  - `SCHED_FIFO` — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
  - `SCHED_RR` — similar to `SCHED_FIFO`, except that threads at the same priority level timeslice (round robin) every `4 × the clock period` (see `ClockPeriod()`).
  - `SCHED_OTHER` — currently the same as `SCHED_RR`.
  - `SCHED_SPORADIC` — sporadic scheduling. For more information, see `pthread_attr_setschedpolicy()`, and “Scheduling algorithms” in the chapter on the Neutrino microkernel in the *System Architecture* guide.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sched_get_priority_max()` function returns the maximum value for the scheduling policy specified by `policy`.

**Returns:**

The appropriate minimum for success, or -1 (`errno` is set).
sched_get_priority_max()

Errors:

EINVAL   The value of the policy parameter doesn’t represent a defined scheduling policy.
ENOSYS   The sched_get_priority_max() function isn’t currently supported.

Examples:

See sched_getparam().

Classification:

POSIX 1003.1 PS

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sched_getparam(), sched_get_priority_min(), sched_setparam(),
sched_getscheduler(), sched_setscheduler()
sched_get_priority_min() © 2007, QNX Software Systems GmbH & Co. KG.
Get the minimum priority for the scheduling policy

Synopsis:

```
#include <sched.h>

int sched_get_priority_min( int policy );
```

Arguments:

- `policy` The scheduling policy, which must be one of:
  - SCHED_FIFO — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
  - SCHED_RR — similar to SCHED_FIFO, except that threads at the same priority level timeslice (round robin) every 4 × the clock period (see `ClockPeriod()`).
  - SCHED_OTHER — currently the same as SCHED_RR.
  - SCHED_Sporadic — sporadic scheduling. For more information, see `pthread_attr_setschedpolicy()`, and “Scheduling algorithms” in the chapter on the Neutrino microkernel in the System Architecture guide.

Library:

```
libc
```

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sched_get_priority_min()` function returns the minimum value for the scheduling policy specified by `policy`.

Returns:

The appropriate minimum for success, or -1 (`errno` is set).
Errors:

- **EINVAL**  The value of the `policy` parameter doesn’t represent a defined scheduling policy.
- **ENOSYS**  The `sched_get_priority_min()` function isn’t currently supported.

Classification:

- **POSIX 1003.1 PS**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `sched_getparam()`, `sched_get_priority_max()`, `sched_setparam()`, `sched_getscheduler()`, `sched_setscheduler()`
**sched_getscheduler()**  © 2007, QNX Software Systems GmbH & Co. KG.

Get the current scheduling policy for a process

**Synopsis:**

```c
#include <sched.h>

int sched_getscheduler( pid_t pid );
```

**Arguments:**

- `pid` The ID of the process whose scheduling policy you want to find, or zero if you want to get the policy for the current process.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sched_getscheduler()` function gets the current scheduling policy of process `pid`. If `pid` is zero, the scheduling policy of the calling process is returned.

**Returns:**

The scheduling policy, or -1 if an error occurred (`errno` is set).

**Errors:**

- `ESRCH` The process `pid` doesn’t exist.

**Classification:**

- POSIX 1003.1 PS
sched_getscheduler()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Currently, the implementation of `sched_getscheduler()` isn’t 100% POSIX 1003.1-1996. The `sched_getscheduler()` function returns the scheduling policy for thread 1 in the process `pid`, or for the calling thread if `pid` is 0.

If you depend on this in new code, it will not be portable. POSIX 1003.1 says `sched_getscheduler()` should return -1 and set `errno` to EPERM in a multithreaded application.

See also:

`errno`, `getprio()`, `sched_getparam()`, `sched_get_priority_max()`,
`sched_get_priority_min()`, `sched_setparam()`, `sched_setscheduler()`,
`sched_yield()`, `setprio()`
sched_param
© 2007, QNX Software Systems GmbH & Co. KG.
Structure that describes scheduling parameters

Synopsis:
#include <sched.h>

struct sched_param {
    int32_t sched_priority;
    int32_t sched_curpriority;
    union {
        int32_t reserved[8];
        struct {
            int32_t __ss_low_priority;
            int32_t __ss_max_repl;
            struct timespec __ss_repl_period;
            struct timespec __ss_init_budget;
        } __ss;
    } __ss_un;
};

#define sched_ss_low_priority __ss_un.__ss.__ss_low_priority
#define sched_ss_max_repl __ss_un.__ss.__ss_max_repl
#define sched_ss_repl_period __ss_un.__ss.__ss_repl_period
#define sched_ss_init_budget __ss_un.__ss.__ss_init_budget

Description:
You’ll use the sched_param structure when you get or set the scheduling parameters for a thread or process.

You can use these functions to get the scheduling parameters:

- pthread_attr_getschedparam()
- pthread_getschedparam()
- sched_getparam()
- SchedGet()

You can use these functions to set the scheduling parameters:

- pthread_attr_setschedparam()
- pthread_setschedparam()
The members of `sched_param` include:

**sched_priority**

When you get the scheduling parameters, this member reflects the priority that was assigned to the thread or process. It doesn’t reflect any temporary adjustments due to priority inheritance.

When you set the scheduling parameters, set this member to the priority that you want to use. The priority must be between the minimum and maximum values returned by `sched_get_priority_min()` and `sched_get_priority_max()` for the scheduling policy.

**sched_curpriority**

When you get the scheduling parameters, this member is set to the priority that the thread or process is currently running at. This is the value that the kernel uses when making scheduling decisions.

When you set the scheduling parameters, this member is ignored.

The other members are used with sporadic scheduling. The following `#define` directives create the POSIX names that correspond to those members and should be used instead of accessing members directly.

**sched_ss_low_priority**

The background or low priority for the thread that’s executing.

**sched_ss_max_repl**

The maximum number of times a replenishment will be scheduled, typically because of a blocking operation. After a
thread has blocked this many times, it automatically drops to
the low-priority level for the remainder of its execution until its
execution budget is replenished.

sched_ss_repl_period

The time that should be used for scheduling the replenishment
of an execution budget after being blocked or having overrun
the maximum number of replenishments. This time is used as
an offset against the time that a thread is made READY.

sched_ss_init_budget

The time that should be used for the thread’s execution budget.
As the thread runs at its high-priority level, its execution time is
carved out of this budget. Once the budget is entirely depleted,
the thread drops to its low-priority level, where, if possible
because of priority arrangements, it can continue to run until the
execution budget is replenished.

- The sched_priority must always be higher than
  sched_ss_low_priority.
- The sched_ss_max_repl must be smaller than SS_REPL_MAX.
- The sched_ss_init_budget must be larger than
  sched_ss_repl_period.

For more information, see “Scheduling algorithms” in the Neutrino
Microkernel chapter of the System Architecture guide.

Examples:

This code shows a duty-cycle usage of the sporadic server thread:

```c
#include <stdio.h>
#include <errno.h>
#include <sched.h>
#include <pthread.h>
#include <inttypes.h>
#include <sys/syspage.h>
```
#include <sys/neutrino.h>

/* 50 % duty cycle of 5 secs on 5 secs off */
struct timespec g_init_budget = { 5, 0 };
struct timespec g_repl_period = { 10, 0 };

#define MY_HIGH_PRIORITY 5
#define MY_LOW_PRIORITY 4
#define MY_REPL_PERIOD g_repl_period
#define MY_INIT_BUDGET g_init_budget
#define MY_MAX_REPL 10
#define DUTY_CYCLE_LOOPS 10

/*
 * Run a compute-bound thread (minimal blocking) to show the
duty cycle.
*/
void *st_duty_check(void *arg) {
    struct sched_param params;
    uint64_t stime, etime, cps;
    double secs;
    int ret, prio;
    int prevprio, iterations;

    stime = ClockCycles();
    cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    iterations = 0;

    printf("\n");
    prevprio = -1;
    while(iterations < DUTY_CYCLE_LOOPS) {
        etime = ClockCycles();
        ret = pthread_getschedparam(pthread_self(), &prio, &params);

        if(ret != 0) {
            printf("pthread_getschedparam() failed %d \n", errno);
            break;
        } else if (prevprio != -1 && prevprio !=
            params.sched_priority) {
            stime = etime - stime;
            secs = (double)stime / (double)cps;
            printf("pri %d (cur %d) %lld cycles %g secs\n",
                    params.sched_priority, 
                    params.sched_curpriority, 
                    stime, secs);
            stime = etime;
        } else {
            printf("%lld\n", secs);
        }
        prevprio = params.sched_priority;
        iterations++;
    }
    etime = ClockCycles();
    printf("%d\n", etime - stime);
    return 0;
}

September 10, 2007
C Library — S 2699
int main(int argc, char **argv) {
    struct sched_param params;
    pthread_attr_t attr;
    pthread_t thr;
    int ret;

    /* Set the attribute structure with the sporadic values */
    printf("# Set sporadic attributes ...");
    pthread_attr_init(&attr);
    ret = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    if(ret != 0) {
        printf("pthread_attr_setinheritsched() failed %d \n", errno);
        return 1;
    }
    ret = pthread_attr_setschedpolicy(&attr, SCHED_SPORADIC);
    if(ret != 0) {
        printf("pthread_attr_setschedpolicy() failed %d %d
", ret, errno);
        return 1;
    }

    params.sched_priority = MY_HIGH_PRIORITY;
    params.sched_ss_low_priority = MY_LOW_PRIORITY;
    memcpy(&params.sched_ss_init_budget, &MY_INIT_BUDGET,
            sizeof(MY_INIT_BUDGET));
    memcpy(&params.sched_ss_repl_period, &MY_REPL_PERIOD,
            sizeof(MY_REPL_PERIOD));
    params.sched_ss_max_repl = MY_MAX_REPL;
    ret = pthread_attr_setschedparam(&attr, &params);
    if(ret != 0) {
        printf("pthread_attr_setschedparam() failed %d \n", errno);
        return 1;
    }
    printf("OK\n");

    /* Create a sporadic thread to check the duty cycle */
    printf("# Creating thread duty cycle thread (%d changes) ...", DUTY_CYCLE_LOOPS);
    ret = pthread_create(&thr, &attr, st_duty_check, NULL);
if(ret != 0) {
    printf("pthread_create() failed %d \n", errno);
    return 1;
}

pthread_join(thr, NULL);
printf("OK\n");

return 0;
}

See also sched_getparam().

Classification:

POSIX 1003.1 PS

See also:

pthread_attr_getschedparam(), pthread_attr_setschedparam(),
pthread_getschedparam(), pthread_setschedparam(),
sched_getparam(), sched_setparam(), sched_setscheduler(),
SchedGet(), SchedSet(), ThreadCreate()

“Scheduling algorithms” in the Neutrino Microkernel chapter of the
System Architecture guide
**sched_rr_get_interval()**  
© 2007, QNX Software Systems GmbH & Co. KG.

*Get the execution time limit of a process*

**Synopsis:**
```
#include <sched.h>

int sched_rr_get_interval(
    pid_t pid,
    struct timespec *interval);
```

**Arguments:**
- **pid**  
The process ID whose execution time limit you want to get.
- **interval**  
A pointer to a `timespec` structure that the function updates with the process’s current execution time limit.

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sched_rr_get_interval()` function updates `interval` with the current execution time limit for the process, `pid`. If `pid` is 0, the current execution time limit for the calling process is returned.

**Returns:**
- **0**  
Success.
- **-1**  
An error occurred (`errno` is set).

**Errors:**
- **ENOSYS**  
The `sched_rr_get_interval()` function isn’t currently supported.
- **ESRCH**  
The process `pid` can’t be found.
**Classification:**

POSIX 1003.1 PS

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

timespec
**sched_setparam()**

Change the priority of a process

**Synopsis:**

```c
#include <sched.h>

int sched_setparam(
    pid_t pid,
    const struct sched_param *param );
```

**Arguments:**

- `pid` The ID of the process whose priority you want to set, or 0 to set it for the current process.
- `param` A pointer to a `sched_param` structure whose `sched_priority` member holds the priority that you want to assign to the process.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sched_setparam()` function changes the priority of process `pid` to that of the `sched_priority` member in the `sched_param` structure pointed to by `param`. If `pid` is zero, the priority of the calling process is changed.

The `sched_priority` member in `param` must lie between the minimum and maximum values returned by `sched_get_priority_max()` and `sched_get_priority_min()`.

By default, the process priority and scheduling algorithm are inherited from or explicitly set by the parent process. Once running, the child process may change its priority by using this function.
sched_setparam()

Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (errno is set).</td>
</tr>
</tbody>
</table>

Errors:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>A fault occurred trying to access the buffers provided.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The priority isn’t a valid priority.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The calling process doesn’t have sufficient privilege to set the priority.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The process pid doesn’t exist.</td>
</tr>
</tbody>
</table>

Classification:

POSIX 1003.1 PS

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Currently, the implementation of sched_setparam() isn’t 100% POSIX 1003.1-1996. The sched_setparam() function sets the scheduling parameters for thread 1 in the process pid, or for the calling thread if pid is 0.

If you depend on this in new code, it will not be portable. POSIX 1003.1 says sched_setparam() should return -1 and set errno to EPERM in a multithreaded application.
See also:

errno, getprio(), sched_getparam(), sched_get_priority_max(),
sched_get_priority_min(), sched_setscheduler(), sched_param,
sched_setscheduler(), sched_yield(), setprio()
**sched_setscheduler()**

*Change the priority and scheduling policy of a process*

**Synopsis:**

```c
#include <sched.h>

int sched_setscheduler(
    pid_t pid,
    int policy,
    const struct sched_param *param);
```

**Arguments:**

- `pid`: The ID of the process whose priority and scheduling policy you want to set, or zero if you want to set them for the current process.
- `policy`: The scheduling policy, which must be one of:
  - `SCHED_FIFO` — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
  - `SCHED_RR` — similar to `SCHED_FIFO`, except that threads at the same priority level timeslice (round robin) every $4 \times$ the clock period (see `ClockPeriod()`).
  - `SCHED_OTHER` — currently the same as `SCHED_RR`.
  - `SCHED_SPORADIC` — sporadic scheduling.

For more information, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

- `param`: A pointer to a `sched_param` structure whose `sched_priority` member holds the priority that you want to assign to the process.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**sched_setscheduler()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The `sched_setscheduler()` function changes the priority of process `pid` to that of the `sched_priority` member in the `sched_param` structure passed as `param`, and the scheduling policy is set to `policy`.

If `pid` is zero, the policy and priority of the calling process are set.

The `sched_priority` member in `param` must lie between the minimum and maximum values returned by `sched_get_priority_max()` and `sched_get_priority_min()`.

By default, process priority and scheduling algorithm are inherited from or explicitly set by the parent process. Once running, the child process may change its priority by using this function.

**Returns:**

The previous scheduling policy, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EFAULT** A fault occurred trying to access the buffers provided.
- **EINVAL** The priority or scheduling policy isn’t a valid value.
- **EPERM** The calling process doesn’t have sufficient privilege to set the priority.
- **ESRCH** The process `pid` doesn’t exist.

**Examples:**

See `sched_getparam()`.

**Classification:**

POSIX 1003.1 PS
sched_setscheduler()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

Currently, the implementation of `sched_setscheduler()` isn’t 100% POSIX 1003.1-1996. The `sched_setscheduler()` function sets the scheduling policy for thread 1 in the process `pid`, or for the calling thread if `pid` is 0.

If you depend on this in new code, it won’t be portable. POSIX 1003.1 says `sched_setscheduler()` should return -1 and set `errno` to EPERM in a multithreaded application.

**See also:**

`errno`, `getprio()`, `sched_getparam()`, `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getscheduler()`, `sched_param()`, `sched_setscheduler()`, `sched_setparam()`, `sched_yield()`, `setprio()`
**sched_yield()**

Yield to other ready threads at the same priority

**Synopsis:**

```c
#include <sched.h>

int sched_yield( void );
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sched_yield()` function checks to see if other threads, at the same priority as that of the calling thread, are READY to run. If so, the calling thread yields to them and places itself at the end of the READY thread queue. The `sched_yield()` function never yields to a lower priority thread.

A higher priority thread always forces a lower priority thread to yield (that is, preempt) the instant the higher priority thread becomes ready to run, without the need for the lower priority thread to give up the processor by calling the `sched_yield()` or `SchedYield()` functions.

The `sched_yield()` function calls the kernel function `SchedYield()`, and may be more portable across realtime POSIX systems.

---

You should avoid designing programs that contain busy wait loops. If you can't avoid them, you can use `sched_yield()` to reduce the system load at a given priority level. Note that a thread that calls `sched_yield()` in a tight loop will spend a great deal of time in the kernel, which will have a small effect on interrupt latency.

---

**Returns:**

This function always succeeds and returns zero.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

int main( void )
{
    int i;

    for( ; ; ) {
        /* Process something... */
        for( i = 0 ; i < 1000 ; ++i )
            fun();

        /* Yield to anyone else at the same priority */
        sched_yield();
    }
    return EXIT_SUCCESS; /* Never reached */
}

int fun()
{
    int i;

    for( i = 0 ; i < 10 ; ++i )
        i += i;

    return( i );
}
```

Classification:

POSIX 1003.1 PS | THR

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

getprio(), sched_getparam(), sched_get_priority_max(),
sched_get_priority_min(), sched_getscheduler(), sched_setparam(),
sched_setscheduler(), SchedYield(), setprio(), sleep()
SchedCtl(), SchedCtl_r()
Control the adaptive partitioning scheduler

Synopsis:

```c
#include <sys/sched_aps.h>

int SchedCtl(int cmd,
              void *data,
              int length);

int SchedCtl_r(int cmd,
                void *data,
                int length);
```

Arguments:

- **cmd** The control command that you want to execute; one of:
  - SCHED_APS_QUERY_PARMS
  - SCHED_APS_SET_PARMS
  - SCHED_APS_CREATE_PARTITION
  - SCHED_APS_LOOKUP
  - SCHED_APS_QUERY_PARTITION
  - SCHED_APS_JOIN_PARTITION
  - SCHED_APS_MODIFY_PARTITION
  - SCHED_APS_PARTITION_STATS
  - SCHED_APS_OVERALL_STATS
  - SCHED_APS_MARK_CRITICAL
  - SCHED_APS_CLEAR_CRITICAL
  - SCHED_APS_ATTACH_EVENTS
  - SCHED_APS_QUERY_THREAD
  - SCHED_APS_ADD_SECURITY
  - SCHED_APS_QUERY_PROCESS

- **data** A pointer to the specific data structure for the command.

- **length** The size of the structure that **data** points to.

For details about each command and its data, see the sections below.
Library:

libc

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

Description:

The \texttt{SchedCtl()} and \texttt{SchedCtl_r()} kernel calls control the adaptive partitioning scheduler.

This scheduler is optional and is present only if you add \texttt{[module=aps]} to your OS image's buildfile. For more information, see the Adaptive Partitioning \textit{User's Guide}. These functions were added in the QNX Neutrino Core OS 6.3.2.

These functions are identical except in the way they indicate errors. See the Returns section for details.

You must initialize all of the fields—including reserved ones—in the structures you pass as the \texttt{data} argument, by calling (for example) \texttt{memset()}. You can also use the \texttt{APS\_INIT\_DATA()} macro:

\begin{verbatim}
APS_INIT_DATA( &data );
\end{verbatim}

\textbf{SCHED\_APS\_QUERY\_PARMS}

This command fills in a \texttt{sched\_aps\_info} structure that describes the overall parameters of the adaptive partitioning scheduler:

\begin{verbatim}
typedef struct {
    _Uint64t cycles_per_ms;
    _Uint64t windowsize_cycles;
    _Uint64t windowsize2_cycles;
    _Uint64t windowsize3_cycles;
    _Uint32t scheduling_policy_flags;
    _Uint32t sec_flags;
    _Uint16t bankruptcy_policy;
    _Uint16t num_partitions;
    _Uint16t max_partitions;
    _Uint64t reserved1;
    _Uint64t reserved2;
} sched_aps_info;
\end{verbatim}
The members include:

\textit{cycles\_per\_ms}

The number of machine cycles in a millisecond. Use this value to convert the output of the \texttt{SCHED\_APS\_QUERY\_PARTITION} command to the time units of your choice.

The value of \textit{cycles\_per\_ms}:

\begin{itemize}
  \item might not equal the value of the \textit{cycles\_per\_sec} member of the system page divided by 1000
  \item isn’t necessarily in the same units as values returned by \texttt{ClockCycles()} on all platforms
\end{itemize}

\textit{windowsize\_cycles}

The length, in CPU cycles, of the averaging window used for scheduling. By default, this corresponds to 100 ms.

If you change the tick size of the system at runtime, do so before defining the adaptive partitioning scheduler’s window size. That’s because Neutrino converts the window size from milliseconds to clock ticks for internal use.

\textit{windowsize2\_cycles} 

The length, in CPU cycles, of window 2, for reporting only. Typically 10 times the window size.

\textit{windowsize3\_cycles} 

The length, in CPU cycles, of window 3, for reporting only. Typically 100 times the window size.

\textit{scheduling\_policy\_flags} 

The set of \texttt{SCHED\_APS\_SCHEDPOL\_} * flags that describe the scheduling policy. For more information, see “Scheduling policies,” below.
sec_flags  The set of SCHED_APS_SEC_* flags that describe the security options. For more information, see “Security,” below.

bankruptcy_policy

What to do if a partition exhausts its critical budget; a combination of SCHED_APS_BNKR_* flags (see “Handling bankruptcy,” below).

num_partitions

The number of partitions defined.

max_partitions

The largest number of partitions that may be created at any time.

Scheduling policies

These flags set options for the adaptive partitioning scheduling algorithm. To set, pass a pointer to an ORed set of these flags with the SCHED_APS_SET_PARMS call to SchedCtl():

SCHED_APS_SCHDPOL_FREETIME_BY_RATIO

Free time is when at least one partition isn’t running. Its time becomes free to other partitions that may then run over their budgets.

By default, the scheduler hands out free time to the partition with the highest-priority running thread. That guarantees realtime scheduling behavior (i.e. scheduling strictly by priority) to partitions any time they aren’t being limited by some other partition’s right to its guaranteed minimum budget. But it also means that one partition is allowed to grab all the free time.

If you set SCHED_APS_SCHDPOL_FREETIME_BY_RATIO, the running partitions share the free time in proportion to the ratios of their budgets. So, one partition can no longer grab all the free time. However, when this flag is set, partitions will see strict priority-scheduling between partitions only when they’re consuming less than their CPU budgets.
SCHED_APS_SCHEDPOL_BMP_SAFETY

Strict priority scheduling between partitions, with some combinations of partition budgets, and some combinations of runmasks (i.e. bound multiprocessing) can require the adaptive partitioning scheduler to not meet minimum CPU budgets. When SCHED_APS_SCHEDPOL_BMP_SAFETY is set, the scheduler uses a more restrictive algorithm that guarantees minimum CPU budgets, but gives priority-based scheduling between partitions only when partitions are consuming less than their budgets.

If this flag is set, SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO is also automatically set.

SCHED_APS_SCHEDPOL_DEFAULT

Neither SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO nor SCHED_APS_SCHEDPOL_BMP_SAFETY. Neutrino sets this at startup.

Scheduling within a partition is always strictly by priority, no matter which of these flags are set.

For more information about adaptive partitioning and BMP, see the Adaptive Partitioning Scheduling Details chapter of the Adaptive Partitioning User’s Guide.

Handling bankruptcy

Bankruptcy is when critical CPU time billed to a partition exceeds its critical budget. Bankruptcy is always considered to be a design error on the part of the application, but you can configure how the system responds to it.

If the system isn’t declaring bankruptcy when you expect it, note that bankruptcy can be declared only if critical time is billed to your partition. Critical time is billed on those timeslices when the following conditions are all met:

- The running partition has a critical budget greater than zero.
The top thread in the partition is marked as running critical, or has received the critical state from receiving a SIG_INTR, a sigevent marked as critical, or has just received a message from a critical thread.

The running partition must be out of percentage-CPU budget.

There be at least one other partition that is competing for CPU time.

Only then if the critical time, billed over the current averaging window, exceeds a partition’s critical budget will the system declare the partition bankrupt.

When the system detects that a partition has gone bankrupt, it always:

- causes that partition to be out-of-budget for the remainder of the current scheduling window

- delivers any sigevent that you’ve specified as a notification of bankruptcy with the SCHED_APS_ATTACH_EVENTS command. This occurs at most once per calling SCHED_APS_ATTACH_EVENTS.

In addition, you can configure the following responses:

**SCHED_APS_BNKR_BASIC**

Deliver bankruptcy-notification events and make the partition out-of-budget for the rest of the scheduling window (nominally 100 ms). This is the default.

**SCHED_APS_BNKR_CANCEL_BUDGET**

Set the offending partition’s critical budget to zero, which forces the partition to be scheduled by its percentage CPU budget only. This also means that a second bankruptcy can’t occur. This persists until a restart occurs, or you call SCHED_APS_MODIFY_PARTITION to set a new critical budget.
SchedCtl(), SchedCtl_r()

SCHED_APS_BNKR_LOG
Not currently implemented.

SCHED_APS_BNKR_REBOOT
Cause the system to crash with a brief message identifying the offending partition. This is the most severe response, suggested for use while testing a product, to make sure bankruptcies are never ignored. You probably shouldn’t use this option in your finished product.

SCHED_APS_BNKR_RECOMMENDED
The combination SCHED_APS_BNKR_CANCEL_BUDGET | SCHED_APS_BNKR_LOG. We recommend this choice.

To set a choice of bankruptcy-handling options, OR the above SCHED_APS_BNKR_* flags and pass a pointer to it as the bankruptcy_policyp field of the sched_aps_parms structure when you call SCHED_APS_SET_PARMS.

Errors:

SchedCtl() and SchedCtl_r() indicate the following errors for the SCHED_APS_QUERY_PARMS command (see the Returns section for details):

EOK Success.

EACCES The calling thread doesn’t meet the security options set (see SCHED_APS_ADD_SECURITY). Usually this means you must be root.

EDOM A reserved field isn’t zero. You might not have used APS_INIT_DATA() to initialize the data parameter.

EINVAL The size of the parameter block doesn’t match the size of the expected structure.

ENOSYS The adaptive partitioning scheduler isn’t installed.
SCHED_APS_SET_PARMS

The command sets the parameters for the overall behavior of the adaptive partitioning scheduler. The data argument must be a pointer to a sched_aps_parms structure:

```c
typedef struct {
    _Int16t windowsize_ms;
    _Int16t reserved1;
    _Uint32t *scheduling_policy_flagsp;
    _Uint32t *bankruptcy_policyp;
    _Int32t reserved2;
    _Int64t reserved3;
} sched_aps_parms;
```

The members include:

- **windowsize_ms**
  - The time over which the scheduler is to average CPU cycles and balance the partitions to their percentage budgets as specified by SCHED_APS_CREATE_PARTITION. If you don’t want to set the window size, set this member to -1.

- **scheduling_policy_flagsp**
  - A pointer to an ORed set of SCHED_APS_SCHEDPOL_* flags that specify the scheduling policy. For more information, see “Scheduling policies,” above. If you don’t want to change the scheduling policy, set this member to NULL.

- **bankruptcy_policyp**
  - A pointer to an ORing of SCHED_APS_BNKR_* flags, as described under “Handling bankruptcy,” above. If you don’t want to change these flags, set this member to NULL.

Errors:

_SchedCtl()_ and _SchedCtl_r()_ indicate the following errors for the SCHED_APS_SET_PARMS command (see the Returns section for details):

- **EOK** Success.
EACCES  SCHED_APS_SEC_PARTITIONS_LOCKED is set, or SCHED_APS_SEC_ROOT0_OVERALL is set and you aren’t running as root in the System partition.

For more information, see “Security,” below.

EDOM  A reserved field isn’t zero. You might not have APS_INIT_DATA() to initialize the data parameter.

EINVAL  The size of the parameter block doesn’t match the size of the expected structure.

ENOSYS  The adaptive partitioning scheduler isn’t installed.

SCHED_APS_CREATE_PARTITION

This command creates a new partition which is considered to be a child of the partition that’s calling SchedCtl(). The system automatically creates a partition called System (the value of APS_SYSTEM_PARTITION_NAME) with an ID of 0.

The data argument for this command must be a pointer to a sched_aps_create_parms structure:

typedef struct {
    /* input parms */
    char *name;
    _Uint16t budget_percent;
    _Int16t critical_budget_ms;
    _Uint32t reserved1;
    _Uint64t reserved2;
    /* output parms */
    _Int16t id;
} sched_aps_create_parms;

The input members include:

name  The name of the new partition. If name is NULL or points to an empty string, SchedCtl() assigns a name, in the form Pa, Pb, Pc, and so on. The name must be no longer than APS_PARTITION_NAME_LENGTH, not including the trailing null character, and can’t include any slashes (/).
**budget_percent**

The percentage CPU budget for the new partition. Budgets given to the new partition are subtracted from the parent partition.

Before creating zero-budget partitions, read the cautions in “Setting budgets for resource managers” in the System Considerations chapter of the Adaptive Partitioning *User’s Guide*.

**critical_budget_ms**

The critical budget, in milliseconds, for the partition, or -1 or 0 if you don’t want the partition to have a critical budget. Critical budgets don’t affect the parent, but are automatically limited to be no bigger than the window size.

The output members include:

**id**

The created partition’s ID number, in the range 0 to the maximum number of partitions – 1 (see the *max_partitions* member of the data from a call to SCHED_APS_QUERY_PARMS. The System partition’s ID number is APS_SYSTEM_PARTITION_ID.

**Errors:**

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the SCHED_APS_CREATE_PARTITION command (see the Returns section for details):

**EOK**

Success.

**EACCES**

SCHED_APS_SEC_PARTITIONS_LOCKED is set, or any of these security conditions are set and not satisfied:

- SCHED_APS_SEC_ROOT_MAKES_PARTITIONS
- SCHED_APS_SEC_SYS_MAKES_PARTITIONS
- SCHED_APS_SEC_NONZERO_BUDGETS
SchedCtl(), SchedCtl_r()

- SCHED_APS_SEC_ROOT MAKES_CRITICAL
- SCHED_APS_SEC_SYS MAKES_CRITICAL

For more information, see “Security,” below.

EDOM A reserved field isn’t zero. You might not have used APS_INIT_DATA() to initialize the data parameter.

EDQUOT The parent partition doesn’t have enough budget.

EEXIST Another partition is already using the given name.

EINVAL The size of the parameter block doesn’t match the size of the expected structure, the partition name is badly formed, or the budget is out of range.

ENAMETOOLONG The partition name is longer than APS_PARTITION_NAME_LENGTH characters.

ENOSPC The maximum number of partitions already exist.

ENOSYS The adaptive partitioning scheduler isn’t installed.

SCHED_APS_QUERY_PARTITION

This command gets information about a given partition. The data argument for this command must be a pointer to a sched_aps_partition_info structure:

typedef struct {
    /* out parms */
    _Uint64t budget_cycles;
    _Uint64t critical_budget_cycles;
    char name[APS_PARTITION_NAME_LENGTH+1];
    _Int16t parent_id;
    _Uint16t budget_percent;
    _Int32t notify_pid;
    _Int32t notify_tid;
    _Uint32t pinfo_flags;
    _Int32t pid_at_last_bankruptcy;
    _Int32t tid_at_last_bankruptcy;
    _Int64t reserved1;
    _Int64t reserved2;
} sched_aps_partition_info;
The input members include:

\textbf{id} \quad \text{The number of the partition you want to query.}

The output members include:

\textbf{budget\_cycles} \quad \text{The budget, in cycles. To convert this value to something useful, convert it with the \textit{cycles\_per\_ms} value from a \textsc{SCHED\_APS\_QUERY\_PARMS} command.}

\textbf{critical\_budget\_cycles} \quad \text{The critical budget, in cycles.}

\textbf{name} \quad \text{The name of the partition.}

\textbf{parent\_id} \quad \text{The number of the partition that’s the parent of the partition being queried. The System partition’s ID number is \textsc{APS\_SYSTEM\_PARTITION\_ID}.}

\textbf{budget\_percent} \quad \text{The partition’s budget, expressed as a percentage.}

\textbf{notify\_pid, notify\_tid} \quad \text{The process and thread IDs of the thread to be given overload and bankruptcy notifications, or -1 if not set.}

\textbf{pinfo\_flags} \quad \text{A set of flag that give extra information about the partition:}

\begin{itemize}
  \item \textsc{SCHED\_APS\_PINFO\_BANKRUPTCY\_NOTIFY\_ARMED} \quad \text{— see \textsc{SCHED\_APS\_ATTACH\_EVENTS}}
\end{itemize}
SchedCtl(), SchedCtl_r()

- \( \text{SCHED\_APS\_INFO\_OVERLOAD\_NOTIFY\_ARMED} \) — see \( \text{SCHED\_APS\_ATTACH\_EVENTS} \)

\( \text{pid\_at\_last\_bankruptcy}, \text{tid\_at\_last\_bankruptcy} \)

The process and thread IDs at the time of the last bankruptcy, or -1 if there wasn’t a previous bankruptcy.

Errors:

\( \text{SchedCtl()} \) and \( \text{SchedCtl\_r()} \) indicate the following errors for the \( \text{SCHED\_APS\_QUERY\_PARTITION} \) command (see the Returns section for details):

- EOK Success.
- EDOM A reserved field isn’t zero. You might not have used \( \text{APS\_INIT\_DATA()} \) to initialize the data parameter.
- EINVAL The size of the parameter block doesn’t match the size of the expected structure.
- ENOSYS The adaptive partitioning scheduler isn’t installed.

**SCHED\_APS\_LOOKUP**

This command finds the partition ID for a given partition name.

The \textit{data} argument for this command must be a \textit{sched\_aps\_lookup\_parms} structure:

```c
typedef struct {
    /* input parms */
    char *name;
    _Int16t reserved1;
    /* output parms */
    _Int16t id;
} sched_aps_lookup_parms;
```

The input members include:
name  The name of the partition

The output members include:

id  The ID number of the partition, if found.

Errors:

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the
SCHED_APS_LOOKUP command (see the Returns section for details):

EOK       Success.
EDOM      A reserved field isn’t zero. You might not have used
          APS_INIT_DATA() to initialize the data parameter.
EINV      The name wasn’t found.

**SCHED_APS_JOIN_PARTITION**

This command makes the thread specified by the given process and
thread IDs becomes a member of the specified partition. This partition
also becomes the thread’s new home partition, i.e. where it returns
after partition inheritance.

The *data* argument for this command must be a pointer to a
sched_aps_join_parms structure:

```c
typedef struct {
    _Int16t   id;
    _Int16t   reserved1;
    _Int32t   pid;
    _Int32t   tid;
    _Int32t   reserved2;
} sched_aps_join_parms;
```

The members include:

id  The ID number of the partition that the thread is to join.
pid, tid  The process and thread IDs of the thread that you want to
join the specified partition:
  • If both pid and tid are zero, the calling thread joins the
    specified partition.
  • If tid is -1, the process with ID pid joins the partition.
    This doesn’t change the partitions that the process’s
    threads are in; it just sets the partition that the threads
    run in when they’re handling a pulse.

Errors:

SchedCtl() and SchedCtl_r() indicate the following errors for the
SCHED_APS_JOIN_PARTITION command (see the Returns section
for details):

EOK               Success.
EACCES            The following security options are set but not satisfied:
  • SCHED_APS_SEC_ROOT_JOINS
  • SCHED_APS_SEC_SYS_JOINS
  • SCHED_APS_SEC_PARENT_JOINS
  • SCHED_APS_SEC_JOIN_SELF_ONLY
    For more information, see “Security,” below.
EDOM             A reserved field isn’t zero. You might not have used
  APS_INIT_DATA() to initialize the data parameter.
EINVAL           The size of the parameter block doesn’t match the size
  of the expected structure, or the partition with the given
  ID doesn’t exist.
ENOSYS           The adaptive partitioning scheduler isn’t installed.
ESRCH            The pid and tid are invalid.
SCHED_APS_MODIFY_PARTITION

This command changes the parameters of an existing partition. If the new budget’s percent value is different from the current, the difference is either taken from, or returned to, the parent partition’s budget. The critical time parameter affects only the chosen partition, not its parent. To change just one of new budget or new critical time, set the other to -1.

- You can’t use this command to modify the System partition’s budget. To increase the size of the System partition, reduce the budget of one of its child partitions.
- Reducing the size of a partition may cause it not to run for the time of an averaging window, as you may have caused it to become temporarily over-budget. However, reducing the critical time doesn’t trigger the declaration of bankruptcy.

The data argument for this command must be a pointer to a sched_aps_modify_parms structure:

typedef struct {
    _Int16t id;
    _Int16t new_budget_percent;
    _Int16t new_critical_budget_ms;
    _Int16t reserved1;
    _Int64t reserved2;
    _Int64t reserved3;
} sched_aps_modifyParms;

The members include:

id        The ID number of the partition.
new_budget_percent
           The new budget for the partition, expressed as a percentage, or -1 if you don’t want to change it.
new_critical_budget_ms

The new critical budget, in milliseconds, for the partition, or -1 if you don’t want to change it. If the critical budget is greater than the window size, it’s considered to be infinite.

Errors:

SchedCtl() and SchedCtl_r() indicate the following errors for the SCHED_APS_MODIFY_PARTITION command (see the Returns section for details):

EOK    Success.
EACCES SCHED_APS_SEC_PARTITIONS_LOCKED is set, or the following security options are set and not satisfied:
   • SCHED_APS_SEC_PARENT_MODIFIES
   • SCHED_APS_SEC_ROOT MAKES_PARTITIONS
   • SCHED_APS_SEC_SYS MAKES_PARTITIONS
   • SCHED_APS_SEC_NONZERO_BUDGETS
   • SCHED_APS_SEC_ROOT MAKES_CRITICAL
   • SCHED_APS_SEC_SYS MAKES_CRITICAL
For more information, see “Security,” below.

EDOM   A reserved field isn’t zero. You might not have used APS_INIT_DATA() to initialize the data parameter.

EINVAL The size of the parameter block doesn’t match the size of the expected structure, or the partition with the given ID doesn’t exist.

ENOSYS The adaptive partitioning scheduler isn’t installed.
SchedCtl(), SchedCtl_r()

SCHED_APS_PARTITION_STATS

This command returns instantaneous values of the CPU time-accounting variables for a set of partitions. It can fill in data for more than one partition. If the length argument to SchedCtl() indicates that you’ve passed the function an array of sched_aps_partition_stats structures, SchedCtl() fills each element with statistics for a different partition, starting with the partition specified by the id field.

To get an accurate picture for the whole machine it’s important to read data for all partitions in one call, since sequential calls to SCHED_APS_PARTITION_STATS may come from separate averaging windows.

To determine the number of partitions, use the SCHED_APS_OVERALL_STATS command.

The command overwrites the id field with the partition number for which data is being returned. It stores -1 into the id field of unused elements.

To convert times in cycles into milliseconds, divide them by the cycles_per_ms obtained with an SCHED_APS_QUERY_PARMS command.

The data argument for this command must be a pointer to a sched_aps_partition_stats structure, or an array of these structures:

```c
typedef struct {
  /* out parms */
  _Uint64t run_time_cycles;
  _Uint64t critical_time_cycles;
  _Uint64t run_time_cycles_w2;
  _Uint64t critical_time_cycles_w2;
  _Uint64t run_time_cycles_w3;
  _Uint64t critical_time_cycles_w3;
  _Uint32t stats_flags;
  _Uint32t reserved1;
  _Uint64t reserved2;
  _Uint64t reserved3;
  /* in parm */
```
The members include:

`run_time_cycles`
The CPU execution time during the last scheduling window.

`critical_time_cycles`
The time spent running critical threads during the last scheduling window.

`run_time_cycles_w2`
The CPU time spent during the last `windowsize2_cycles`. Window 2 is typically 10 times the length of the averaging window.

`critical_time_cycles_w2`
The time spent running critical threads during the last `windowsize2_cycles`.

`run_time_cycles_w3`
The CPU time spent during the last `windowsize3_cycles`. Window 3 is typically 100 times the length of the averaging window.

`critical_time_cycles_w3`
The time spent running critical threads during the last `windowsize3_cycles`.

`stats_flags` A set of the following flags:

- `SCHED_APS_PSTATS_IS_BANKRUPT_NOW` — the critical time used is greater than the critical budget at the time you used the `SCHED_APS_PARTITION_STATS` command.
- `SCHED_APS_PSTATS_WAS_BANKRUPT` — the partition was declared bankrupt sometime since the last restart.
id

This is both an input and output field. As input, it’s the ID number of the first partition you want data for. If you’ve passed an array of `sched_aps_partition_stats` structures, the command fills in the ID number for each partition that it fills in statistics for.

Errors:

`SchedCtl()` and `SchedCtl_r()` indicate the following errors for the SCHED_APS_PARTITION_STATS command (see the Returns section for details):

- **EOK**: Success.
- **EDOM**: A reserved field isn’t zero. You might not have used `APS_INIT_DATA()` to initialize the data parameter.
- **EINVAL**: The size of the parameter block isn’t a multiple of `size(sched_aps_partition_stats)`.
- **ENOSYS**: The adaptive partitioning scheduler isn’t installed.

SCHED_APS_OVERALL_STATS

This command returns instantaneous values of overall CPU-usage variables and other dynamic scheduler states. The `data` argument for this command must be a pointer to a `sched_aps_overall_stats` structure:

```c
typedef struct {
    _Uint64t idle_cycles;
    _Uint64t idle_cycles_w2;
    _Uint64t idle_cycles_w3;
    _Int16t id_at_last_bankruptcy;
    _Uint16t reserved1;
    _Int32t pid_at_last_bankruptcy;
    _Int32t tid_at_last_bankruptcy;
    _Uint32t reserved2;
    _Uint32t reserved3;
    _Uint64t reserved4;
} sched_aps_overall_stats;
```
The members include:

\[ \text{idle\_cycles} \]

The time, in cycles, during the last scheduling window where nothing (other than the idle thread) ran. To convert this to the percent idle time, calculate:

\[
\frac{100 \times \text{idle\_cycles}}{\text{windowsize\_cycles}}
\]

\[ \text{idle\_cycles\_w2} \]

The time spent running idle during the last \( \text{windowsize2\_cycles} \). Window 2 is typically 10 times the length of the averaging window.

\[ \text{idle\_cycles\_w3} \]

The time spent running idle during last \( \text{windowsize3\_cycles} \). Window 3 is typically 100 times the length of the averaging window.

\[ \text{id\_at\_last\_bankruptcy} \]

The ID of last bankrupt partition, or -1 if no bankruptcy has occurred.

\[ \text{pid\_at\_last\_bankruptcy, tid\_at\_last\_bankruptcy} \]

The process and thread IDs at last the bankruptcy, or -1 if no bankruptcy has occurred.

Errors:

\text{SchedCtl()}\text{ and SchedCtl\_r()}\text{ indicate the following errors for the SCHED\_APS\_OVERALL\_STATS command (see the Returns section for details):}\n
\text{EOK} \quad \text{Success.}\n
\text{EDOM} \quad \text{A reserved field isn’t zero. You might not have used APS\_INIT\_DATA() to initialize the data parameter.}\n
\text{EINVAL} \quad \text{The size of the parameter block doesn’t match the size of the expected structure.}\n
\text{ENOSYS} \quad \text{The adaptive partitioning scheduler isn’t installed.}\n
SchedCtl(), SchedCtl_r() © 2007, QNX Software Systems GmbH & Co. KG.

SCHED_APS_MARK_CRITICAL

This command sets one thread in your process to run as a critical thread whenever it runs. Use a thread ID of zero to set the calling thread to be critical.

In general, it’s more useful to send a critical sigevent structure to a thread to make it run as a critical thread.

The data argument for this command must be a pointer to a sched_aps_mark_crit_parms structure:

```c
typedef struct {
    _Int32t pid;
    _Int32t tid;
    _Int32t reserved;
} sched_aps_mark_crit_parms;
```

The members include:

- **pid**: The process ID, or 0 for the calling process.
- **tid**: The thread ID, or 0 for the calling thread.

You can also set up sigevent structures that make their receiving threads run as critical.

Errors:

SchedCtl() and SchedCtl_r() indicate the following errors for the SCHED_APS_MARK_CRITICAL command (see the Returns section for details):

- **EOK**: Success.
- **EDOM**: A reserved field isn’t zero. You might not have used APS_INIT_DATA() to initialize the data parameter.
- **EINVAL**: The size of the parameter block doesn’t match the size of the expected structure.
ENOSYS The adaptive partitioning scheduler isn’t installed.

ESRCH The specified thread wasn’t found.

**SCHED_APS_CLEAR_CRITICAL**

This command clears the “always run as critical” state set by the SCHED_APS_CLEAR_CRITICAL command. Then the thread will run as critical only when it inherits that state from another thread (on receipt of a message).

The data argument for this command must be a pointer to a `sched_aps_clear_crit_parms` structure:

```c
typedef struct {
    Int32 pid;
    Int32 tid;
    Int32 reserved;
} sched_aps_clear_crit_parms;
```

The members include:

- **pid** The process ID, or 0 for the calling process.
- **tid** The thread ID, or 0 for the calling thread.

**Errors:**

*SchedCtl() and SchedCtl_r() indicate the following errors for the SCHED_APS_CLEAR_CRITICAL command (see the Returns section for details):*

- **EOK** Success.
- **EDOM** A reserved field isn’t zero. You might not have used `APS_INIT_DATA()` to initialize the data parameter.
- **EINVAL** The size of the parameter block doesn’t match the size of the expected structure.
- **ENOSYS** The adaptive partitioning scheduler isn’t installed.
- **ESRCH** The specified thread wasn’t found.
SCHED_APS_QUERY_THREAD

This command determines the partition for the given thread and indicates whether or not the thread in your process is marked to run as critical. Use a thread ID of zero to indicate the calling thread.

The data argument for this command must be a pointer to a sched_aps_query_thread_parms structure:

```c
typedef struct {
    _Int32t pid;
    _Int32t tid;
    /* out parms: */
    _Int16t id;
    _Int16t inherited_id;
    _Uint32t crit_state_flags;
    _Int32t reserved1;
    _Int32t reserved2;
} sched_aps_query_thread_parms;
```

The input members include:

- **pid** The ID of process that the thread belongs to, or 0 to indicate the calling process.

- **tid** The thread ID, or 0 for the calling thread.

The output members include:

- **id** The ID number of the partition that the thread originally joined.

- **inherited_id** The ID number of the partition that the thread currently belongs to. This might not be the same as the id member, because the thread might have inherited the partition from a calling process.

- **crit_state_flags** A combination of the following flags:
  - APS_QCRIT_PERM_CRITICAL — the thread always runs as critical.
SchedCtl(), SchedCtl_r()

- **APS_QCRIT_RUNNING_CRITICAL** — the thread is currently running as critical.
- **APS_QCRIT_BILL_AS_CRITICAL** — the thread’s execution time is being billed to the partition’s critical budget.

If **APS_QCRIT_PERM_CRITICAL** isn’t set, and **APS_QCRIT_RUNNING_CRITICAL** is set, it means the thread has temporarily inherited the critical state. If **APS_QCRIT_RUNNING_CRITICAL** is set, and **APS_QCRIT_BILL_AS_CRITICAL** isn’t set, it means that the thread is running as critical, but isn’t depleting its partition’s critical-time budget (i.e. it’s running for free).

Errors:

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the SCHED_APS_QUERY_THREAD command (see the Returns section for details):

- **EOK** Success.
- **EDOM** A reserved field isn’t zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.
- **EINVAL** The size of the parameter block doesn’t match the size of the expected structure.
- **ENOSYS** The adaptive partitioning scheduler isn’t installed.
- **ESRCH** The specified thread wasn’t found.

SCHED_APS_ATTACH_EVENTS

This command defines **sigevent** structures that the scheduler will return to the calling thread when the scheduler detects that a given partition has become bankrupt, or the whole system has become overloaded.
Overload notification isn’t implemented in this release.

Calling SCHED_APS.Attach_EVENTS arms the notification once. After you receive the notification, you must call SCHED_APS.Attach_EVENTS again to receive a subsequent notification. This is to ensure that the system doesn’t send you notifications faster than you can handle them. The pinfo_flags field of the sched_aps_partition_stats structure (see the SCHED_APS_PARTITION_STATS command) indicates if these events are armed.

You can register only one pair of sigevent structures (bankruptcy and overload) per partition, and the notifications must go to the same thread. The thread notified is the calling thread. Attaching events a second time overwrites the first. Passing NULL pointers means “no changes in notification.” To turn off notification, use SIGEV_NONE_INIT() to set the appropriate sigevent to SIGEV_NONE.

If you want to configure additional actions for the system to perform on bankruptcy, see “Handling bankruptcy,” below.

The data argument for this command must be a pointer to a sched_aps_events_parm structure:

```c
typedef struct {
    const struct sigevent  *bankruptcy_notification;
    const struct sigevent  *overload_notification;
    /* each partition gets a different set of sigevents */
    _Int16t id;
    _Int16t reserved1;
    _Int32t reserved2;
    _Int64t reserved3;
} sched_aps_events_parm;
```

The members include:
bankruptcy_notification

A pointer to the sigevent to send to the calling thread if the partition becomes bankrupt, or NULL if you don’t want to change the notification.

overload_notification

Not implemented.

id

The ID of the partition that you want to attach events to, or -1 for the partition of the calling thread. The command updates this member to indicate the partition that it attached the events to.

Errors:

SchedCtl() and SchedCtl_r() indicate the following errors for the SCHED_APS_ATTACH_EVENTS command (see the Returns section for details):

EOK Success.

EACCES You don’t have the right to modify the partition, i.e the following security modes are set and not satisfied:

- SCHED_APS_SEC_PARENT_MODIFIES
- SCHED_APS_SEC_ROOT MAKES_PARTITIONS
- SCHED_APS_SEC_SYS MAKES_PARTITIONS

For more information, see “Security,” below.

EDOM A reserved field isn’t zero. You might not have used APS_INIT_DATA() to initialize the data parameter.

EINVAL The size of the parameter block doesn’t match the size of the expected structure.

ENOSYS The adaptive partitioning scheduler isn’t installed.

ESRCH The specified thread wasn’t found.
SCHED_APS_ADD_SECURITY

This command sets security options. A bit that’s set turns the corresponding security option on. Successive calls add to the existing set of security options. Security options can only be cleared by a restart.

You must be root running in the System partition to use this command, even if all security options are off.

The data argument for this command must be a pointer to a sched_aps_security_parms structure:

typedef struct {
    _Uint32t     sec_flags;
    _Uint32t     reserved1;
    _Uint32t     reserved2;
} sched_aps_security_parms;

The members include:

sec_flags A set of SCHED_APS_SEC_* flags (see below), as both input and output parameters. Set this member to 0 if you want to get the current security flags.

Security

The adaptive partitioning scheduler lets you dynamically create and modify the partitions in your system.

We recommend that you set up your partition environment at boot time, and then lock all parameters:

- in a program, by using the SCHED_APS_SEC_LOCK_PARTITIONS flag
- from the command line, by using the aps modify command

However you might need to modify a partition at runtime. In this case, you can use the security options described below.
When Neutrino starts, it sets the security option to APS_SCHED_SEC_OFF. We recommend that you immediately set it to SCHED_APS_SEC_RECOMMENDED. In code, do this:

```c
sched_aps_security_parms p;
APS_INIT_DATA( &p );
p.sec_flags = SCHED_APS_SEC_RECOMMENDED;
SchedCtl( SCHED_APS_ADD_SECURITY,&p, sizeof(p) );
```

These are the security options:

**SCHED_APS_SEC_RECOMMENDED**

Only `root` from the System partition may create partitions or change parameters. This arranges a 2-level hierarchy of partitions: the System partition and its children. Only `root`, running in the System partition, may join its own thread to partitions. The percentage budgets must not be zero.

**SCHED_APS_SEC_FLEXIBLE**

Only `root` in the System partition can change scheduling parameters or change critical budgets. But `root` running in any partition can create subpartitions, join threads into its own subpartitions and modify subpartitions. This lets applications create their own local subpartitions out of their own budgets. The percentage budgets must not be zero.

**SCHED_APS_SEC_BASIC**

Only `root` in the System partition may change overall scheduling parameters and set critical budgets.

Unless you’re testing the partitioning and want to change all parameters without needing to restart, you should set at least SCHED_APS_SEC_BASIC.

In general, SCHED_APS_SEC_RECOMMENDED is more secure than SCHED_APS_SEC_FLEXIBLE, which is more secure than SCHED_APS_SEC_BASIC. All three allow partitions to be created and modified. After setting up partitions, use...
SchedCtl(), SchedCtl_r() © 2007, QNX Software Systems GmbH & Co. KG.

SCHED_APS_SEC_LOCK_PARTITIONS to prevent further unauthorized changes. For example:

```c
sched_aps_securityParms p;
APS_INIT_DATA(&p);
p.sec_flags = SCHED_APS_SEC_LOCK_PARTITIONS;
SchedCtl(SCHED_APS_ADD_SECURITY, &p, sizeof(p));
```

SCHED_APS_SEC_RECOMMENDED, SCHED_APS_SEC_FLEXIBLE, and SCHED_APS_SEC_BASIC are composed of the flags defined below (but it’s usually more convenient for you to use the compound options):

**SCHED_APS_SEC_ROOT0_OVERALL**

You must be root running in the System partition in order to change the overall scheduling parameters, such as the averaging window size.

**SCHED_APS_SEC_ROOT MAKES PARTITIONS**

You must be root in order to create or modify partitions. Applies to the SCHED_APS_CREATE_PARTITION, SCHED_APS_MODIFY_PARTITION, and SCHED_APS_ATTACH_EVENTS commands.

**SCHED_APS_SEC_SYS MAKES PARTITIONS**

You must be running in the System partition in order to create or modify partitions. This applies to same commands as SCHED_APS_SEC_ROOT MAKES PARTITIONS. Attaching events, with SCHED_APS_ATTACH_EVENTS, is considered to be modifying the partition.

**SCHED_APS_SEC_PARENT_MODIFIES**

Allows partitions to be modified (SCHED_APS_MODIFY_PARTITION), but you must be running in the parent partition of the partition being modified. “Modify” means to change a partition’s percentage or critical budget or attach events with the SCHED_APS_ATTACH_EVENTS command.
SCHED_APS_SEC_NONZERO_BUDGETS

A partition may not be created with, or modified to have, a zero budget. Unless you know that all your partitions need to run only in response to client requests, i.e. receipt of messages, you should set this option.

SCHED_APS_SEC_ROOT_MAKES_CRITICAL

You have to be root in order to create a nonzero critical budget or change an existing critical budget.

SCHED_APS_SEC_SYS_MAKES_CRITICAL

You must be running in the System partition to create a nonzero critical budget or change an existing critical budget.

SCHED_APS_SEC_ROOT_JOINS

You must be root in order to join a thread to a partition.

SCHED_APS_SEC_SYS_JOINS

You must be running in the System partition in order to join a thread.

SCHED_APS_SEC_PARENT_JOINS

You must be running in the parent partition of the partition you wish to join to.

SCHED_APS_SEC_JOIN_SELF_ONLY

The caller of the SCHED_APS_JOIN_PARTITION command must specify 0 for the pid and tid. In other words, a process may join only itself to a partition.

SCHED_APS_SEC_PARTITION_LOCKED

Prevent further changes to any partition’s budget, or overall scheduling parameters, such as the window size. Set this after you’ve set up your partitions. Once you’ve locked the partitions, you can still use the SCHED_APS_JOIN_PARTITION and SCHED_APS_ATTACH_EVENTS commands.
Errors:

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the \*SCHED_\*APS_\*ADD_\*SECURITY\* command (see the Returns section for details):

- **EOK** Success.
- **EACCES** The calling thread doesn’t meet the security options set (see \*SCHED_\*APS_\*ADD_\*SECURITY\*). Usually this means you must be \*root\*.
- **EDOM** A reserved field isn’t zero. You might not have used \*APS_\*INIT_\*DATA()\* to initialize the data parameter.
- **EINV AL** The size of the parameter block doesn’t match the size of the expected structure.
- **ENOSYS** The adaptive partitioning scheduler isn’t installed.

**SCHED_\*APS_\*QUERY_\*PROCESS**

This command returns the partition of the given process. The partition of a process is billed while one of the process’s threads handles a pulse. The individual threads in a process may all be in different partitions from the process.

The *data* argument for this command must be a pointer to a

```
typedef struct {
    _Int32t    pid;
    /* out parms: */
    _Int16t    id;  /* partition of process */
    _Int16t    reserved1;
    _Int32t    reserved2;
    _Int32t    reserved3;
    _Int32t    reserved4;
} sched_aps_query_process_parms;
```

The members include:

- **\*pid** The process ID, or 0 for the calling process.
- **\*id** The ID of the process’s partition.
Errors:

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the SCHED_APS_QUERY_PROCESS command (see the Returns section for details):

- **EOK**: Success.
- **EDOM**: A reserved field isn’t zero. You might not have used *APS_INIT_DATA()* to initialize the data parameter.
- **EINVAL**: The size of the parameter block doesn’t match the size of the expected structure.
- **ENOSYS**: The adaptive partitioning scheduler isn’t installed.
- **ESRCH**: The process wasn’t found.

Blocking states

This call doesn’t block.

Returns:

The only difference between these functions is the way they indicate errors:

- **SchedCtl()**: EOK if the call succeeds. If an error occurs, *SchedCtl()* returns -1 and sets *errno*.
- **SchedCtl_r()**: EOK if the call succeeds. This function **doesn’t** set *errno*. If an error occurs, *SchedCtl_r()* returns the negative of an error value.

For a list of error codes, see the description of each command.
Examples:

```c
sched_aps_partition_info part_info;

// You need to initialize the parameter block.
APS_INIT_DATA(&part_info);

// Set the input members of the parameter block.
part_info.id = 2;

// Invoke SchedCtl to query the partition.
ret = SchedCtl( SCHED_APS_QUERY_PARTITION, &part_info,
                sizeof(part_info) );
if (EOK!=ret) some_kind_of_error_handler();

// Use output field
printf( "The budget is %d per cent.\n",
        part_info.budget_percent);
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `SchedGet()`, `SchedInfo()`, `SchedSet()`, `SchedYield()`, `sigevent`
- `aps` in the Utilities Reference
- Adaptive Partitioning User's Guide
Synopsis:

```c
#include <sys/neutrino.h>

int SchedGet( pid_t pid,
               int tid,
               struct sched_param *param );

int SchedGet_r( pid_t pid,
                int tid,
                struct sched_param *param );
```

Arguments:

- `pid` 0 or a process ID; see below.
- `tid` 0 or a thread ID; see below.
- `param` A pointer to a `sched_param` structure where the function can store the scheduling parameters.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SchedGet()` and `SchedGet_r()` kernel calls return the current scheduling policy and the parameters for the thread specified by `tid` in the process specified by `pid`. If `pid` is zero, the current process is used to look up a nonzero `tid`. If `pid` and `tid` are zero, then the calling thread is used.

These functions are identical except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling `pthread_getschedparam()`. The scheduling policy is returned on success and is one of SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, SCHED_OTHER, SCHED_ADJTOHEAD, or SCHED_ADJTOTAIL.

**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

- `SchedGet()` The current scheduling policy. If an error occurs, -1 is returned and `errno` is set.

- `SchedGet_r()` The current scheduling policy. This function does NOT set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

**Errors:**

- EFAULT A fault occurred when the kernel tried to access the buffers provided.

- ESRCH The process indicated by `pid` or thread indicated by `tid` doesn’t exist.

**Classification:**

QNX Neutrino
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_getschedparam()`, `sched_param`, `_schedInfo()`, `SchedSet()`, `SchedYield()`
Get scheduler information

**Synopsis:**
```c
#include <sys/neutrino.h>

int SchedInfo( pid_t pid,
    int policy,
    struct _sched_info* info );

int SchedInfo_r( pid_t pid,
    int policy,
    struct _sched_info* info );
```

**Arguments:**
- `pid` A process ID, or 0 to get information about the current process.
- `policy` One of the following:
  - `SCHED_FIFO` — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher-priority thread.
  - `SCHED_RR` — the same as `SCHED_FIFO`, except threads at the same priority level timeslice (round robin) every $4 \times$ the clock period (see `ClockPeriod()`).
  - `SCHED_OTHER` — currently the same as `SCHED_RR`.
  - `SCHED_SPORADIC` — sporadic scheduling. For more information, see `pthread_attr_setschedpolicy()`, and “Scheduling algorithms” in the chapter on the Neutrino microkernel in the System Architecture guide.
- `info` A pointer to a `_sched_info` structure where the function can store the scheduler information.

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

These kernel calls return information about the kernel’s thread scheduler, including the minimum and maximum thread priority, for the process ID specified by `pid` when using the specified scheduling `policy`. If `pid` is 0, the scheduler information for the current process is returned. In either case, the `struct _sched_info` pointed to by `info` is filled in with the appropriate information.

The `SchedInfo()` and `SchedInfo_r()` functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `sched_get_priority_max()`, `sched_get_priority_min()`, or `sched_rr_get_interval()`.

The `struct _sched_info` structure pointed to by `info` contains at least these members:

- `uint64_t interval`
  The current execution time limit before the thread is suspended in favor of the scheduler.

- `int priority_max`
  The maximum priority for a thread using this scheduling `policy`.

- `int priority_min`
  The minimum priority for a thread using this scheduling `policy`.

Returns:

The only difference between these functions is the way they indicate errors:

- `SchedInfo()`  If an error occurs, -1 is returned and `errno` is set. Any other value returned indicates success.
- `SchedInfo_r()`  EOK is returned on success. This function does NOT set `errno`. If an error occurs, any value in the Errors section may be returned.
SchedInfo(), SchedInfo_r() © 2007, QNX Software Systems GmbH & Co. KG.

Errors:

EINV AL The pid or policy is invalid.
ENOSYS The SchedInfo() function isn’t supported by this system.
ESRCH The process specified by pid doesn’t exist.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sched_get_priority_max(), sched_get_priority_min(),
sched_rr_get_interval(), SchedGet(), SchedSet()
Set the scheduling policy for a thread

Synopsis:

```c
#include <sys/neutrino.h>

int SchedSet(
    pid_t pid,
    int tid,
    int policy,
    const struct sched_param *param );

int SchedSet_r(
    pid_t pid,
    int tid,
    int policy,
    const struct sched_param *param );
```

Arguments:

- `pid` 0 or a process ID; see below.
- `tid` 0 or a thread ID; see below.
- `policy` The scheduling policy; one of:
  - `SCHED_FIFO` — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher priority thread.
  - `SCHED_RR` — the same as `SCHED_FIFO`, except threads at the same priority level timeslice (round robin) every 4 × the clock period (see `ClockPeriod()`).
  - `SCHEDOTHER` — currently the same as `SCHED_RR`.
  - `SCHED_OTHER` — currently the same as `SCHED_RR`.
  - `SCHED_SPORADIC` — sporadic scheduling.
  - `SCHED_NOCHANGE` — this isn’t actually a policy, but a special value that tells the kernel to update the parameters specified in `param`, without changing the policy.
  - `SCHED_ADJTOHEAD` — puts `pid` and `tid` at the head of the READY queue if they are in a READY state.
SchedSet(), SchedSet_r()

- SCHED_ADJTOTAIL — puts pid and tid at the tail of the READY queue if they are in a READY state.

If either the SCHED_ADJTOHEAD or SCHED_ADJTOTAIL options is set, then the struct sched_param field is ignored.

For more information, see “Thread scheduling” in the QNX Neutrino Microkernel chapter of the System Architecture guide.

param A pointer to a sched_param structure where the function can store the scheduling parameters.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The SchedSet() and SchedSet_r() kernel calls set both the scheduling policy and the associated parameters for the thread specified by tid in the process specified by pid. If pid is zero the current process is used to look up a nonzero tid. If tid is zero, then the calling thread is used and pid is ignored.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling pthread_setschedparam().

Blocking states

These calls don’t block.
**SchedSet(), SchedSet_r()**

**Returns:**

The only difference between these functions is the way they indicate errors:

- **SchedSet()**
  
  If an error occurs, -1 is returned and *errno* is set. Any other value returned indicates success.

- **SchedSet_r()**
  
  EOK is returned on success. This function does NOT set *errno*. If an error occurs, any value in the Errors section may be returned.

**Errors:**

- **EFAULT**
  
  A fault occurred when the kernel tried to access the buffers you provided.

- **EINVAL**
  
  The given scheduling policy is invalid.

- **EPERM**
  
  The process doesn’t have permission to change the scheduling of the indicated thread.

- **ESRCH**
  
  The process indicated by *pid* or thread indicated by *tid* doesn’t exist.

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`pthread_setschedparam()`, `sched_get_priority_max()`,
`sched_get_priority_min()`, `sched_param`, `SchedSet()`, `SchedInfo()`,
`SchedYield()`
Synopsis:

#include <sys/neutrino.h>

int SchedYield( void );
int SchedYield_r( void );

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

These kernel calls check to see if other threads at the same priority as that of the calling thread are ready to run. If so, the calling thread yields to them and places itself at the end of the ready thread queue for that priority. SchedYield() never yields to a lower priority thread. Higher priority threads always force a yield the instant they become ready to run. This call has no effect with respect to threads running at priorities other than the calling thread’s.

The SchedYield() and SchedYield_r() functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling sched_yield().

Avoid designing programs that contain busy-wait loops using SchedYield() to timeslice. If this is unavoidable, you can use SchedYield() to reduce the system load at a given priority level. Note that a program that calls SchedYield() in a tight loop will spend a great deal of time in the kernel, which will have a small effect on scheduling interrupt latency.
**Blocking states**

These calls don’t block. However, if other threads are ready at the same priority, the calling thread is placed at the end of the ready queue for this priority.

**Returns:**

The only difference between these functions is the way they indicate errors:

*SchedYield()*. If an error occurs, -1 is returned and *errno* is set. Any other value returned indicates success.

*SchedYield_r()*. EOK is returned on success. This function does **NOT** set *errno*. If an error occurs, any value in the Errors section may be returned.

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*_sched_yield(), SchedGet(), SchedSet()*
Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>

int socket( PF_INET,
            SOCK_DGRAM,
            IPPROTO_SCTP);
```

Or,

```c
int socket( PF_INET,
            SOCK_STREAM,
            IPPROTO_SCTP);
```

Description:

The SCTP protocol provides a reliable, end-to-end message transport service. It has the following features:

- acknowledged error-free nonduplicated transfer of user data
- data fragmentation to conform to path MTU size
- sequenced delivery of user messages within multiple streams
- multi-homing
- protection against Denial of Service (DOS) attacks.

Returns:

A descriptor referencing the socket, or -1 if an error occurs (errno is set).

Errors:

- EACCES Permission to create a socket of the specified type and/or protocol is denied.
EMFILE The per-process descriptor table is full.
ENFILE The system file table is full.
ENOBUFFS Insufficient buffer space available. The socket can’t be created until sufficient resources are freed.
ENOMEM Not enough memory.
EPROTONOSUPPORT

The protocol type or the specified protocol isn’t supported within this domain.

See also:

IP

sctp_bindx(), sctp_connectx(), sctp_freeladdrx(), sctp_freepaddrx(),
sctp_getladdrx(), sctp_getpaddrx(), sctp_peelloff(), sctp_recvmsg(),
sctp_sendmsg()

RFC 2960, RFC 3257

Drafts:

- Socket API extension for stream control transmission protocol in accord with draft-ietf-tsvwg-sctpsocket-07.txt.
- Stream control transmission protocol dynamic address reconfiguration.
sctp_bindx()

Add or remove one or more given addresses from an association

Synopsis:

```c
#include <netinet/sctp.h>

int sctp_bindx(int sd,
               struct sockaddr *addrs,
               int addrcnt,
               int flags);
```

Arguments:

- **sd**: Socket descriptor. Depending on the type of `sd`, the type of address is determined. If `sd` is an IPv4 socket, the address passed is an IPv4 address. If `sd` is an IPv6 socket, the address passed is either an IPv4 or an IPv6 address. A single address is specified as INADDR_ANY or IN6ADDR_ANY.

- **addrs**: A pointer to an array of one or more socket addresses. Each address is contained in its appropriate structure (i.e. `struct sockaddr_in` or `struct sockaddr_in6`). The family of the address type must be used to distinguish the address length. This representation is termed a “packed array” of addresses.

- **addrcnt**: Number of addresses in the array.

- **flags**: Either SCTP_BINDX_ADD_ADDR or SCTP_BINDX_REM_ADDR.

Library:

- **libsctp**

Use the `-l sctp` option to `qcc` to link against this library.
**sctp_bindx()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The *sctp_bindx()* function adds or removes one or more addresses to or from an association:

- Use SCTP_BINDX_ADD_ADDR to associate additional addresses with an endpoint after calling *bind()*.

- Use SCTP_BINDX_REM_ADDR to remove some addresses that a listening socket is associated with, so that no new association will be associated with those addresses.

If the endpoint supports dynamic addressing, adding or removing an address may cause an endpoint to send the appropriate message to change the peer’s address lists.

**Returns:**

0  Success.

-1  Failure; *errno* is set.

**Errors:**

EFAULT  Passed-in flag was neither SCTP_BINDX_ADD_ADDR nor SCTP_BINDX_REM_ADDR.

EINVAL  Passed-in address has a wrong family.

**Classification:**

SCTP

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

*continued…*
sctp_bindx()

Safety
---
Thread: Yes

See also:

SCTP, sctp_connectx(), sctp_freeladdr(), sctp_freepaddr(),
sctp_getladdr(), sctp_getpaddr(), sctp_peeloff(), sctp_recvmsg(),
sctp_sendmsg()
**sctp_connectx()**

Connect a host to a multihomed endpoint

**Synopsis:**

```c
#include <netinet/sctp.h>

int sctp_connectx( int s, 
    struct sockaddr *addrs, 
    int addrcnt);
```

**Arguments:**

- `s`  Socket descriptor.
- `addrs`  Array of addresses.
- `addrcnt`  Number of addresses in the array.

**Library:**

`libsctp`

Use the `-l sctp` option to `qcc` to link against this library.

**Description:**

The `sctp_connectx()` function connects a host to a multihomed endpoint by specifying a list of peer addresses.

**Returns:**

- `0`  Success.
- `-1`  Failure; `errno` is set.

**Errors:**

- `EINVAL`  An invalid address was passed in `addrs`. 
**sctp_connectx()**

**Classification:**

SCTP

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

SCTP, `sctp_bindx()`, `sctp_freeladdr()`, `sctp_freepaddr()`, `sctp_getladdr()`, `sctp_getpaddr()`, `sctp_peeloff()`, `sctp_recvmsg()`, `sctp_sendmsg()`
**sctp_freeladdr()**

Free all resources allocated by `sctp_getladdrs()`

### Synopsis:
```
#include <netinet/sctp.h>

void sctp_freeladdrs(struct sockaddr *addrs);
```

### Arguments:
- `addrs` Array of local addresses returned by `sctp_getladdrs()`.

### Library:
- `libsctp`

Use the `-l sctp` option to `qcc` to link against this library.

### Description:
The `sctp_freeladdrs()` free all resources allocated by `sctp_getladdrs()`.

### Classification:
- **SCTP**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:
- SCTP, `sctp_bindx()`, `sctp_connectx()`, `sctp_freepaddrs()`, `sctp_getladdrs()`, `sctp_getpaddr()`, `sctp_peeloff()`, `sctp_recvmsg()`, `sctp_sendmsg()`
Synopsis:

```
#include <netinet/sctp.h>

void sctp_freepaddrs(struct sockaddr *addrs);
```

Arguments:

- `addrs` Array of peer addresses returned by `sctp_getpaddrs()`.

Library:

- `libscpt`

Use the `-l scpt` option to `qcc` to link against this library.

Description:

The `sctp_freepaddrs()` free all resources allocated by `sctp_getpaddrs()`.

Classification:

- SCTP

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
sctp_freepaddrs()

See also:

SCTP, sctp_bindx(), sctp_connectx(), sctp_freeladdrx(),
sctp_getladdrx(), sctp_getpaddrx(), sctp_peeloff(), sctp_recvmsg(),
sctp_sendmsg()
**sctp_getladdrs()**

Get all locally bound addresses on a socket

**Synopsis:**

```c
#include <netinet/sctp.h>

int sctp_getladdrs(int sd,
                    sctp_assoc_t id,
                    struct sockaddr **addrs);
```

**Arguments:**

- **sd**: Socket descriptor.
- **id**: Specifies the association for one-to-many style sockets. It is ignored for one-to-one style sockets.
- **addrs**: A pointer to an array of local addresses, returned by the stack.

**Library:**

`libsctp`

Use the `-l sctp` option to `qcc` to link against this library.

**Description:**

The `sctp_getladdrs()` function gets all locally bound addresses on a socket.

On return, `addrs` points to a dynamically allocated packed array of `sockaddr` structures of the appropriate type for each address. Use `sctp_freepaddrs()` to free the memory. Note that the in-and-out parameter `addrs` must not be NULL.

**Returns:**

On success, `sctp_getladdrs()` returns the number of local addresses bound to the socket. If the socket is unbound, `sctp_getladdrs()` returns 0, and the value of `*addrs` is undefined.
If an error occurs, `sctp_getladdrs()` returns -1, and the value of *addrs is undefined.

**Errors:**

- **EINVAL** — The address is invalid.
- **ENOTCONN** — The socket isn’t bound.
- **ENOMEM** — Can’t allocate memory for the array of addresses.

**Classification:**

- **SCTP**

  **Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- SCTP, `sctp_bindx()`, `sctp_connectx()`, `sctp_freeladdr()`, `sctp_freepaddr()`, `sctp_getpaddrs()`, `sctp_peeloff()`, `sctp_recvmsg()`, `sctp_sendmsg()`
**Synopsis:**

```c
#include <netinet/sctp.h>

int sctp_getpaddrs(int sd,
                   sctp_assoc_t id,
                   struct sockaddr **addrs);
```

**Arguments:**

- `sd` Socket descriptor.
- `id` Specifies the association for one-to-many style sockets. It is ignored for one-to-one style sockets
- `addrs` A pointer to an array of peer addresses, returned by the stack.

**Library:**

```bash
libsctp
```

Use the `-l sctp` option to `qcc` to link against this library.

**Description:**

The `sctp_getpaddrs()` function gets all peer addresses in an association.

On return, `addrs` points to a dynamically packed array of `sockaddr` structures of the appropriate type for each address. Use `sctp_freepaddrs()` to free the memory. Note that the in-and-out parameter `addrs` must not be NULL.

**Returns:**

On success, `sctp_getpaddrs()` returns the number of peer addresses in the association. If there is no association, this function returns 0 and the value of `*addrs` is undefined.
If an error occurs, `sctp_getpaddrs()` returns -1, and the value of *addrs* is undefined.

**Errors:**

- EINVAL: The passed-in address is invalid.
- ENOMEM: Can’t allocate memory for the array of addresses.

**Classification:**

SCTP

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

SCTP, `sctp_bindx()`, `sctp_connectx()`, `sctp_freeladdr()`, `sctp_freepaddr()`, `sctp_getladdr()`, `sctp_peeloff()`, `sctp_recvmsg()`, `sctp_sendmsg()`
**Synopsis:**

```c
#include <netinet/sctp.h>

int sctp_peeloff(int sd,
                  sctp_assoc_t assoc_id);
```

**Library:**

```c
libsctp
```

Use the `-l sctp` option to `qcc` to link against this library.

**Description:**

You call this function to branch off an association into a separate socket. The new socket is a one-to-one style socket. You should confine your operation to one that is allowed for a one-to-one style socket.

Using `sctp_peeloff()`, you create a new socket descriptor as follows:

```c
new_sd = sctp_peeloff(int sd, sctp_assoc_t assoc_id);
```

**Returns:**

On success, it returns a new socket, which has the single association in it. On failure, it returns -1 and `errno` is set.

**Errors:**

- **EBADF**   The socket descriptor, `sd`, is invalid.
- **ENOTSOCK**   Can’t branch off the association.
## sctp_peeloff()

### Classification:

SCTP

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

SCTP, `sctp_bindx()`, `sctp_connectx()`, `sctp_freeladdr()`, `sctp_freepaddr()`, `sctp_getladdr()`, `sctp_getpaddr()`, `sctp_recvmsg()`, `sctp_sendmsg()`
sctp_recvmsg()

Synopsis:

```c
#include <netinet/sctp.h>

ssize_t sctp_recvmsg(int s,
        void *msg,
        size_t len,
        struct sockaddr *from,
        socklen_t *fromlen,
        struct sctp_sndrcvinfo *sinfo,
        int *msg_flags);
```

Arguments:

- `s`  
  Socket descriptor.

- `msg`  
  Message buffer to be filled.

- `len`  
  Length of the message buffer.

- `from`  
  A pointer to a `sockaddr` object where the function can store the source address of the message.

- `fromlen`  
  A pointer to a `socklen_t` object that specifies the size of the `from` buffer. The function stores the actual size of the address in this object.

- `sinfo`  
  A pointer to a `sctp_sndrcvinfo` structure to be filled upon receipt of the message.

- `msg_flags`  
  A pointer to an integer to be filled with any message flags (e.g. MSG_NOTIFICATION).

Library:

```bash
libsctp
```

Use the `-l sctp` option to `qcc` to link against this library.
**sctp_recvmsg()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Description:**

The `sctp_recvmsg()` function receives a message from socket `s`, whether or not a socket is connected. The difference between this function and the generic function, `recvmsg()`, is that you can pass in a pointer to a `sctp_sndrcvinfo` structure, and the structure is filled upon receipt of the message. The structure has detailed information about the message you just received.

You must enable `sctp_data_io_events` with the SCTP_EVENTS socket option first, to be able to have the `sctp_sndrcvinfo` structure be filled in.

**Returns:**

The number of bytes received, or -1 if an error occurs (`errno` is set).

**Errors:**

ENOMEM Not enough stack memory.

**Classification:**

SCTP

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

SCTP, `sctp_bindx()`, `sctp_connectx()`, `sctp_freeladdr()`,
`sctp_freepaddr()`, `sctp_getladdr()`, `sctp_getpaddr()`, `sctp_peeloff()`,
`sctp_sendmsg()`
sctp_sendmsg() © 2007, QNX Software Systems GmbH & Co. KG.
Send a message, using advanced SCTP features

Synopsis:
```
#include <netinet/sctp.h>

ssize_t sctp_sendmsg(int s,
    const void *msg,
    size_t len,
    struct sockaddr *to,
    socklen_t tolen,
    uint32_t ppid,
    uint32_t flags,
    uint16_t stream_no,
    uint32_t timetolive,
    uint32_t context);
```

Arguments:
- **s**: Socket descriptor.
- **msg**: Message to be sent.
- **len**: Length of the message.
- **to**: Destination address of the message.
- **tolen**: Length of the destination address.
- **ppid**: An opaque unsigned value that is passed to the remote end in each user message. The byte order issues are not accounted for and this information is passed opaquely by the SCTP stack from one end to the other.
- **flags**: Flags composed of bitwise OR of these values:
  - **MSG_UNORDERED**: This flag requests the unordered delivery of the message. If the flag is clear, the datagram is considered an ordered send.
MSG_ADDR_OVER
This flag, in one-to-many style, requests the SCTP stack to override the primary destination address.

MSG_ABORT
This flag causes the specified association to abort — by sending an ABORT message to the peer (one-to-many style only).

MSG_EOF
This flag invokes the SCTP graceful shutdown procedures on the specified association. Graceful shutdown assures that all data enqueued by both endpoints is successfully transmitted before closing the association (one-to-many style only).

stream_no
Message stream number — for the application to send a message. If a sender specifies an invalid stream number, an error indication is returned and the call fails.

timetolive
Message time to live in milliseconds. The sending side expires the message within the specified time period if the message has not been sent to the peer within this time period. This value overrides any default value set using socket option. If you use a value of 0, it indicates that no timeout should occur on this message.

context
An opaque 32-bit context datum. This value is passed back to the upper layer if an error occurs while sending a message, and is retrieved with each undelivered message.

Library:

libsctp
Use the -l sctp option to qcc to link against this library.
**Description:**

The `sctp_sendmsg()` function allows you to send extra information to a remote application. Using advanced SCTP features, you can send a message through a specified stream, pass extra opaque information to a remote application, or define a timeout for the particular message.

**Returns:**

The number of bytes sent, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EBADF** An invalid descriptor was specified.
- **EDESTADDRREQ** A destination address is required.
- **EFAULT** An invalid user space address was specified for a parameter.
- **EMSGSIZE** The socket requires that the message be sent atomically, but the size of the message made this impossible.
- **ENOBUFS** The system couldn’t allocate an internal buffer. The operation may succeed when buffers become available.
- **ENOTSOCK** The argument `s` isn’t a socket.
- **EWOULDBLOCK** The socket is marked nonblocking and the requested operation would block.

**Classification:**

SCTP
sctp_sendmsg()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

SCTP, sctp_bindx(), sctp_connectx(), sctp_freeladdr(), sctp_freepaddr(), sctp_getladdr(), sctp_getpaddr(), sctp_peeloff(), sctp_recvmsg(), sctp_sendmsg().
searchenv()

Search the directories listed in an environment variable

Synopsis:

```c
#include <stdlib.h>

void searchenv( const char* name,  
                const char* env_var,  
                char* buffer );
```

Arguments:

- `name` The name of the file that you want to search for.
- `env_var` The name of an environment variable whose value is a list of directories that you want to search. Common values for `env_var` are "PATH", "LIB" and "INCLUDE".

The `searchenv()` function doesn’t search the current directory unless it’s specified in the environment variable.

- `buffer` A buffer where the function can store the full path of the file found. This buffer should be PATH_MAX bytes long. If the specified file can’t be found, the function stores an empty string in the buffer.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `searchenv()` function searches for the file specified by `name` in the list of directories assigned to the environment variable specified by `env_var`.
searchenv()

Use pathfind() or pathfind_r() instead of this function.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void display_help( FILE *fp )
{
    printf( "display_help T.B.I.\n" );
}

int main( void )
{
    FILE *help_file;
    char full_path[ PATH_MAX ];

    searchenv( "lib_ref.html", "PATH", full_path );
    if( full_path[0] == '\0' ) {
        printf( "Unable to find help file\n" );
    } else {
        help_file = fopen( full_path, "r" );
        display_help( help_file );
        fclose( help_file );
    }

    return EXIT_SUCCESS;
}
```

Classification:

QNX 4

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

September 10, 2007
Caveats:

The `searchenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

`getenv()`, `pathfind()`, `pathfind_r()`, `setenv()`
Synopsis:

```c
#include <stdlib.h>

unsigned short int *seed48(
    unsigned short int seed16v[3] );
```

Arguments:

`seed16v` An array that comprises the 48 bits of the seed.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `seed48()` initializes the internal buffer `r(n)` of `drand48()`, `lrand48()`, and `mrand48()`. All 48 bits of the seed can be specified in an array of 3 short integers, where the entry with index 0 specifies the lowest bits. The constant multiplicand and addend of the algorithm are reset to the defaults: the multiplicand `a = 0xFDEECE66D = 25214903917` and the addend `c = 0xB = 11`.

Returns:

A pointer to an array of 3 shorts which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to `seed48()`.

Classification:

POSIX 1003.1 XSI
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `mrand48()`, `nrand48()`, `srand48()`
set the position for the next read of the directory stream

Synopsis:

```c
#include <dirent.h>

void seekdir( DIR * dirp,
              long int pos );
```

Arguments:

- `dirp` A pointer to the directory stream, for which you want to set the current location.
- `pos` The new position for the directory stream. You should have obtained this value from an earlier call to `telldir()`.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `seekdir()` function sets the position of the next `readdir()` operation on the directory stream specified by `dirp` to the position specified by `pos`.

The new position reverts to the one associated with the directory stream when the `telldir()` operation was performed.

Values returned by `telldir()` are good only for the lifetime of the `DIR` pointer, `dirp`, from which they’re derived. If the directory is closed and then reopened, the `telldir()` value may be invalidated due to undetected directory compaction. It’s safe to use a previous `telldir()` value immediately after a call to `opendir()` and before any calls to `readdir()`. 
seekdir()

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

closedir(), errno, lstat(), opendir(), readdir(), readdir_r(),
rewinddir(), telldir(), stat()
select()  
Check for files that are ready for reading or writing

Synopsis:

```
#include <sys/select.h>

int select( int width,
           fd_set * readfds,
           fd_set * writefds,
           fd_set * exceptfds,
           struct timeval * timeout );
```

```
FD_SET( int fd, fd_set * fdset );
FD_CLR( int fd, fd_set * fdset );
FD_ISSET( int fd, fd_set * fdset );
FD_ZERO( fd_set * fdset );
```

Arguments:

- **width**  
The number of descriptors to check in the given sets. Only the descriptors from 0 through (width-1) in the descriptor sets are examined. Therefore, the value of width must be at least as large as:

  
  (highest valued file descriptor in the sets) + 1

- **readfds**  
  NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for reading. The function replaces the set with the file descriptors that are actually ready for reading.

- **writefds**  
  NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that are ready for writing. The function replaces the set with the file descriptors that are actually ready for writing.

- **exceptfds**  
  NULL, or a pointer to a `fd_set` object that specifies the descriptors to check for files that have an exceptional condition pending. The function replaces the set with the file descriptors that actually have an exceptional condition pending.

- **timeout**  
  NULL, or a pointer to a `timeval` that specifies how long to wait for the selection to complete.
select()

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The select() function examines the file descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending. Any of readfds, writefds, and exceptfds may be NULL pointers if no descriptors are of interest.

In earlier versions of QNX Neutrino, select() and the associated macros were defined in sys/time.h. They’re now defined in sys/select.h, which sys/time.h includes.

The select() function replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation, and returns the total number of ready descriptors in all the sets.

If timeout isn’t NULL, it specifies a maximum interval to wait for the selection to complete. If timeout is NULL, select() blocks until one of the selected conditions occurs. To effect a poll, the timeout argument should be a non-NULL pointer, pointing to a zero-valued timeval structure.

If the current operating system configuration supports a larger number of open files than is specified in FD_SETSIZE, you can increase the number of open file descriptors used with select() by changing the definition of FD_SETSIZE before including <sys/select.h> or <sys/time.h>.

If you use select() with a timeout, you should reset the timeout value after calling select().
If you’re using `select()` in conjunction with the socket API package, note that selecting for reading on a socket descriptor on which a `listen()` has been performed indicates that a subsequent `accept()` on that descriptor won’t block.

### Manipulating file-descriptor sets

At least the following macros are defined in `<sys/select.h>` for manipulating file-descriptor sets:

- `FD_ZERO( &fdset )`
  - Initialize a descriptor set `fdset` to the null set.

- `FD_SET( fd, &fdset )`
  - Add the file descriptor `fd` to the set `fdset`.

- `FD_CLR( fd, &fdset )`
  - Remove `fd` from `fdset`.

- `FD_ISSET( fd, &fdset )`
  - Is nonzero if `fd` is a member of `fdset`; otherwise, zero.

The behavior of these macros is undefined if a descriptor value is less than zero, or greater than or equal to `FD_SETSIZE`.

### Returns:

The number of ready descriptors in the descriptor sets, 0 if the `timeout` expired, or -1 if an error occurs (`errno` is set).

### Errors:

- **EBADF**  One of the descriptor sets specified an invalid descriptor.
- **EFAULT** One of the pointers given in the call referred to a nonexistent portion of the address space for the process.
EINTR     A signal was delivered before any of the selected events occurred, or before the time limit expired.

EINVAL    A component of the pointed-to time limit is outside the acceptable range: \( t_{\text{sec}} \) must be between 0 and \( 10^8 \), inclusive; \( t_{\text{usec}} \) must be greater than or equal to 0, and less than \( 10^6 \).

Examples:

```c
/*
* This example opens a console and a serial port for read mode, and calls select() with a 5 second timeout.
* It waits for data to be available on either descriptor.
*/

#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/select.h>

int main( void )
{
    int console, serial;
    struct timeval tv;
    fd_set rfd;
    int n;

    if( ( console = open( "/dev/con1", O_RDONLY ) ) == -1
        || ( serial = open( "/dev/ser1", O_RDONLY ) ) == -1 )
    {
        perror( "open" );
        return EXIT_FAILURE;
    }

    /* Clear the set of read file descriptors, and add the two we just got from the open calls. */
    FD_ZERO( &rfd );
    FD_SET( console, &rfd );
    FD_SET( serial, &rfd );

    /* Set a 5 second timeout. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;
```
switch ( n = select( 1 + max( console, serial ),
                &rfd, 0, 0, &tv ) ) {
        case -1:
            perror( "select" );
            return EXIT_FAILURE;
        case 0:
            puts( "select timed out" );
            break;
        default:
            printf( "%d descriptors ready ...\n", n );
            if( FD_ISSET( console, &rfd ) )
                puts( " -- console descriptor has data pending" );
            if( FD_ISSET( serial, &rfd ) )
                puts( " -- serial descriptor has data pending" );
        }
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

Cancellation point  No
Interrupt handler  No
Signal handler  No
Thread  Read the Caveats

Caveats:

The select() function only works with raw file descriptors; it doesn't
work with file descriptors in edited mode. See the ICANON flag in the
description of the tcgetattr() function.

The select() function is thread safe as long as the fd sets used by each
thread point to memory that is specific to that thread.
select()

In Neutrino, if multiple threads block in `select()` on the same `fd` for the same condition, all threads may unblock when the condition is satisfied. This may differ from other implementations where only one thread may unblock.

See also:

`errno`, `fcntl()`, `read()`, `sysconf()`, `tcsetattr()`, `write()`
Synopsis:

#include <sys/iofunc.h>
#include <sys/dispatch.h>

int select_attach(
    void *dpp,
    select_attr_t *attr,
    int fd,
    unsigned flags,
    int (*func)( select_context_t *ctp,
        int fd,
        unsigned flags,
        void *handle ),
    void *handle );

Arguments:

dpp The dispatch handle, as returned by dispatch_create(), that you want to attach to a file descriptor.

attr A pointer to a select_attr_t structure. This structure is defined as:

typedef struct _select_attr {
    unsigned flags;
} select_attr_t;

Currently, no attribute flags are defined.

fd The file descriptor that you want to attach to the dispatch handle.

flags Flags that specify the events that you're interested in. For more information, see “Flags,” below.

func The function that you want to call when the file descriptor unblocks. For more information, see “Function,” below.

handle A pointer to arbitrary data that you want to pass to func.
select_attach()

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `select_attach()` attaches the file descriptor `fd` to the dispatch handle `dpp` and selects `flags` events. When `fd` “unblocks”, `func` is called with `handle`.

Flags

The available flags are defined in `<sys/dispatch.h>`. The following flags use `ionotify()` mechanisms (see `ionotify()` for further details):

**SELECT_FLAG_EXCEPT**

Out-of-band data is available. The definition of out-of-band data depends on the resource manager.

**SELECT_FLAG_READ**

There’s input data available. The amount of data available defaults to 1. For a character device such as a serial port, this is a character. For a POSIX message queue, it’s a message. Each resource manager selects an appropriate object.

**SELECT_FLAG_WRITE**

There’s room in the output buffer for more data. The amount of room available needed to satisfy this condition depends on the resource manager. Some resource managers may default to an empty output buffer, while others may choose some percentage of the empty buffer.

These flags are specific to dispatch:

**SELECT_FLAG_REARM**

Rearm the `fd` after an event is dispatched.
SELECT_FLAG_SRVEXCEPT

Register a function that’s called whenever a server, to which this client is connected, dies. (This flag uses the `ChannelCreate()` function’s `_NTO_CHF_COID_DISCONNECT` flag. In this case, `fd` is ignored.)

Function

The argument `func` is the user-supplied function that’s called when one of the registered events occurs on `fd`. This function should return 0 (zero); other values are reserved. The function is passed the following arguments:

- `ctp` Context pointer.
- `fd` The `fd` on which the event occurred.
- `flags` The type of event that occurred. The possible `flags` are:
  - `SELECT_FLAG_except`
  - `SELECT_FLAG_READ`
  - `SELECT_FLAG_WRITE`
  
  For descriptions of the flags passed to `func`, see “Flags,” above.

- `handle` The `handle` passed to `select_attach()`.

Returns:

Zero on success, or -1 if an error occurred (`errno` is set).

Errors:

- `EINVAL` Invalid argument.
- `ENOMEM` Insufficient memory was available.
select_attach()

Examples:

For an example with select_attach(), see dispatch_create(). For other examples using the dispatch interface, see message_attach(), resmgr_attach(), and thread_pool_create().

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

select_detach(), select_query()
select_detach()

Detach a file descriptor from a dispatch handle

Synopsis:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int select_detach( void *dpp,
    int fd );
```

Arguments:

- `dpp` The dispatch handle, as returned by `dispatch_create()`, that you want to detach from the file descriptor.
- `fd` The file descriptor that you want to detach.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The function `select_detach()` detaches the file descriptor `fd` that was registered with dispatch `dpp`, using the `select_attach()` call.

Returns:

- `0` Success.
- `-1` The file descriptor `fd` wasn’t registered with the dispatch `dpp`.

Examples:

```
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int my_func( ... ) {
    ...
}
```
int main( int argc, char **argv ) {
    dispatch_t *dpp;
    int fd;
    select_attr_t attr;

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \n dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    if( argc ≤ 2 || ( fd = open( argv[1],
        O_RDWR | O_NONBLOCK ) ) == -1 ) {
        exit(0);
    }

    select_attach( dpp, &attr, fd,
        SELECT_FLAG_READ | SELECT_FLAG_REARM, my_func, NULL );

    if ( (select_detach( dpp, fd )) == -1 ) {
        fprintf( stderr, "Failed to detach \n the file descriptor.\n" );
        return 1;
    }
}

For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued...*
select_detach()
**select_query()**

*Decode the last select event*

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int select_query
    ( select_context_t *ctp,
      int *fd,
      unsigned *flags,
      int (**func)( select_context_t *ctp,
                   int fd,
                   unsigned flags,
                   void *handle ),
      void **handle );
```

**Arguments:**

- **ctp**
  A pointer to a `select_context_t` structure that defines the context of the event that you want to get information about.

- **fd**
  A pointer to a location where the function can store the file descriptor that’s associated with the event.

- **flags**
  A pointer to a location where the function can store the flags associated with the event; see “Flags” in the documentation for `select_attach()`.

- **func**
  A pointer to a location where the function can store the function associated with the event; see “Function” in the documentation for `select_attach()`.

- **handle**
  A pointer to a location where the function can store the address of any data that you arranged to pass to **func**.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
select_query()

Description:

The function select_query() stores the values of the last select event, for context ctp, in fd, flags, func, and handle.

Returns:

If an error occurs, the function returns -1. An error occurs if the received event doesn’t belong to one of the file descriptors attached with select_attach().

Examples:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int my_func( select_context_t *ctp, 
    int fd, 
    unsigned flags, 
    void *handle ) {

    ...
}

int main( int argc, char **argv ) {
    dispatch_t  *dpp;
    dispatch_context_t  *ctp;
    int        fd;
    unsigned   flag;
    void       *handle;
    select_attr_t  *attr;
    int (*func)( select_context_t *,
                int, unsigned, void * );

    if( ( dpp = dispatch_create() ) == NULL ) {
        fprintf( stderr, "%s: Unable to allocate \
                 dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    if( argc ≤ 2 || (fd = open( argv[1], 
                            O_RDWR | O_NONBLOCK )) == -1 ) {
        exit(0);
    }
```
select_query()

```c
select_attach( dpp, attr, fd,
       SELECT_FLAG_READ | SELECT_FLAG_REARM, &my_func, NULL );

ctp = dispatch_context_alloc( dpp );

if( select_query( (select_context_t *)ctp, &fd, &flag,
          &func, &handle ) == -1 ) {
    fprintf( stderr, "Failed to decode last select event.\n" );
    return 1;
}
```

For more examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`select_attach()`, `select_detach()`
sem_close()

Close a named semaphore

Synopsis:

#include <semaphore.h>

int sem_close( sem_t * sem );

Arguments:

sem A pointer to a semaphore, as returned by sem_open().

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The sem_close() function closes the named semaphore sem opened by sem_open(), releasing any system resources associated with the sem.

Don’t mix named semaphore operations (sem_open() and sem_close()) with unnamed semaphore operations (sem_init() and sem_destroy()) on the same semaphore.

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EINVAL Invalid semaphore descriptor sem.
sem_close()

Classification:

POSIX 1003.1 SEM

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sem_init(), sem_open(), sem_unlink()

proc* in the Utilities Reference
sem_destroy()

Destroy a semaphore

Synopsis:

```c
#include <semaphore.h>

int sem_destroy( sem_t * sem );
```

Arguments:

- `sem` A pointer to the `sem_t` object for the semaphore that you want to destroy.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_destroy()` function destroys the unnamed semaphore referred to by the `sem` argument. The semaphore must have been previously initialized by the `sem_init()` function.

The effect of using a semaphore after it has been destroyed is undefined. If you destroy a semaphore that other processes are currently blocked on, they’re unblocked, with an error (EINVAL).

Don’t mix named semaphore operations (`sem_open()`) and `sem_close()`) with unnamed semaphore operations (`sem_init()`) and `sem_destroy()`) on the same semaphore.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).
**sem_destroy()**

Errors:

- **EINVAL** Invalid semaphore descriptor *sem*.

Classification:

POIX 1003.1 SEM

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `sem_init()`, `sem_post()`, `sem_trywait()`, `sem_wait()`
sem_getvalue()

Get the value of a named or unnamed semaphore

Synopsis:

```c
#include <semaphore.h>

int sem_getvalue( sem_t * sem,
                  int * value );
```

Arguments:

- `sem` A pointer to the `sem_t` object for the semaphore whose value you want to get.
- `value` A pointer to a location where the function can store the semaphore’s value. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a `value` of 0 (zero) indicates a locked semaphore.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_getvalue()` function takes a snapshot of the value of the semaphore, `sem`, and stores it in `value`. This value can change at any time, and is guaranteed valid only at some point in the `sem_getvalue()` call.

This function is provided for debugging semaphore code.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).
sem_getvalue()

Errors:

EINVAL  Invalid semaphore descriptor sem.

Classification:

POSIX 1003.1 SEM

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sem_destroy(), sem_init(), sem_trywait(), sem_wait()
sem_init()
Initialize a semaphore

Synopsis:
#include <semaphore.h>

int sem_init( sem_t * sem, 
  int pshared, 
  unsigned value );

Arguments:

sem A pointer to the sem_t object for the semaphore that you
want to initialize.

pshared Nonzero if you want the semaphore to be shared between
processes via shared memory.

value The initial value of the semaphore. A positive value (i.e.
greater than zero) indicates an unlocked semaphore, and
a value of 0 (zero) indicates a locked semaphore. This
value must not exceed SEM_VALUE_MAX.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The sem_init() function initializes the unnamed semaphore referred to
by the sem argument. The initial counter value of this semaphore is
specified by the value argument.

You can use the initialized semaphore in subsequent calls to
sem_wait(), sem_trywait(), sem_post(), and sem_destroy(). An
initialized semaphore is valid until it's destroyed by the sem_destroy()
function, or until the memory where the semaphore resides is
released.

If the pshared argument is nonzero, then the semaphore can be shared
between processes via shared memory. Any process can then use sem
with the `sem_wait()`, `sem_trywait()`, `sem_post()` and `sem_destroy()` functions.

Don’t mix named semaphore operations (`sem_open()`) and `sem_close()`) with unnamed semaphore operations (`sem_init()` and `sem_destroy()`) on the same semaphore.

**Returns:**

0  Success. The semaphore referred to by `sem` is initialized.

-1  An error occurred (`errno` is set).

**Errors:**

- **EAGAIN**  A resource required to initialize the semaphore has been exhausted.
- **EINVAL**  The `value` argument exceeds `SEM_VALUE_MAX`.
- **EPERM**  The process lacks the appropriate privileges to initialize the semaphore.
- **ENOMEM**  A resource required to initialize the semaphore has been exhausted.
- **ENOSYS**  The `sem_init()` function isn’t supported.

**Classification:**

POSIX 1003.1 SEM

**Safety**

Cancelling point  No
Interrupt handler  No
Signal handler    No

continued...
Safety

Thread: Yes

Caveats:

Don’t initialize the same semaphore from more than one thread. It’s best to set up semaphores before starting any threads.

See also:

`errno`, `sem_destroy()`, `sem_post()`, `sem_trywait()`, `sem_wait()`
sem_open()  © 2007, QNX Software Systems GmbH & Co. KG.
Create or access a named semaphore

Synopsis:

```
#include <semaphore.h>

sem_t * sem_open( const char * sem_name,  
                 int oflags,  
                 ... );
```

Arguments:

- **sem_name** The name of the semaphore that you want to create or access; see below.
- **oflags** Flags that affect how the function creates a new semaphore. This argument is a combination of:
  - O_CREAT
  - O_EXCL

Don’t set **oflags** to O_RDONLY, O_RDWR, or O_WRONLY. A semaphore’s behavior is undefined with these flags. The QNX libraries silently ignore these options, but they may reduce your code’s portability.

For more information, see below.

If you set O_CREAT in **oflags**, you must also pass the following arguments:

- **mode_t mode** The semaphore’s mode (just like file modes). For portability, you should set the read, write, and execute bits to the same value. An easy way of doing this is to use the constants from `<sys/stat.h>`:
  - S_IRWXG for group access.
  - S_IRWXO for other’s access.
  - S_IRWXU for your own access.
sem_open()  

For more information, see “Access permissions” in the documentation for stat().

unsigned int value

The initial value of the semaphore. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a value of 0 (zero) indicates a locked semaphore. This value must not exceed SEM_VALUE_MAX.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The sem_open() function creates or accesses a named semaphore. Named semaphores are slower than the unnamed semaphores created with sem_init(). Semaphores persist as long as the system is up.

If you want to use named semaphores, the named-semaphore manager must be running. Starting with release 6.3.0, procnto, manages named semaphores, which mqueue used to do (and still does, if it detects that procnto isn’t doing so).

The sem_open() function returns a semaphore descriptor that you can use with sem_wait(), sem_trywait(), and sem_post(). You can use it until you call sem_close().

The sem_name argument is interpreted as follows:
sem_open() 

<table>
<thead>
<tr>
<th>name</th>
<th>Pathname space entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>CWD/entry</td>
</tr>
<tr>
<td>/entry</td>
<td>/dev/sem/entry</td>
</tr>
<tr>
<td>entry/newentry</td>
<td>CWD/entry/newentry</td>
</tr>
<tr>
<td>/entry/newentry</td>
<td>/entry/newentry</td>
</tr>
</tbody>
</table>

where CWD is the current working directory for the program at the point that it calls sem_open().

If you want to create or access a semaphore on another node, you have to specify the name as /net/node/sem_location.

The oflags argument is used only for semaphore creation. When creating a new semaphore, you can set oflags to O_CREAT or (O_CREAT | O_EXCL):

O_CREAT Create a new named semaphore. If you set this bit, you must provide the mode and value arguments to sem_open().

O_EXCL When creating a new named semaphore, O_EXCL causes sem_open() to fail if a semaphore with sem_name already exists. Without O_EXCL, sem_open() attaches to an existing semaphore or creates a new one if sem_name doesn’t exist.

Don’t mix named semaphore operations (sem_open() and sem_close()) with unnamed semaphore operations (sem_init() and sem_destroy()) on the same semaphore.
sem_open()

Returns:

A pointer to the created or accessed semaphore, or -1 for failure (errno is set).

Errors:

EACCES Either the named semaphore exists and you don’t have permission to access it, or you’re trying to create a new semaphore and you don’t have permission.

EEXIST You specified O_CREAT and O_EXCL in oflags, but the semaphore already exists.

EINVAL The sem_name argument is invalid or, when creating a semaphore, value is greater than SEM_VALUE_MAX.

EINVAL The call was interrupted by a signal.

EMFILE The process is using too many files or semaphores.

ENFILE The system ran out of resources and couldn’t open the semaphore.

ENAMETOOLONG

The sem_name argument is longer than (NAME_MAX - 8).

ENOENT Either the manager for named semaphores (mqueue, if procnto isn’t managing them) isn’t running, or the sem_name argument doesn’t exist and you didn’t specify O_CREAT in oflags.

ENOSPC There’s insufficient space to create a new named semaphore.

ENOSYS The sem_open() function isn’t implemented for the filesystem specified in sem_name.
Classification:

POSIX 1003.1 SEM

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`sem_close()`, `sem_destroy()`, `sem_init()`, `sem_post()`, `sem_trywait()`,
`sem_unlink()`, `sem_wait()

mqueue, procnto* in the Utilities Reference
Synopsis:

#include <semaphore.h>

int sem_post( sem_t * sem );

Arguments:

sem   A pointer to the sem_t object for the semaphore whose value you want to increment.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The sem_post() function increments the semaphore referenced by the sem argument. If any processes are currently blocked waiting for the semaphore, then one of these processes will return successfully from its call to sem_wait.

The process to be unblocked is determined in accordance with the scheduling policies in effect for the blocked processes. The highest priority waiting process is unblocked, and if there is more than one highest priority process blocked waiting for the semaphore, then the highest priority process that has been waiting the longest is unblocked.

The sem_post() function is reentrant with respect to signals, and can be called from a signal handler.

Returns:

0   Success.
-1  An error occurred (errno is set).
**sem_post()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Errors:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Invalid semaphore descriptor sem.</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>The <code>sem_post()</code> function isn’t supported.</td>
</tr>
</tbody>
</table>

**Classification:**

POSIX 1003.1 SEM

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `sem_destroy()`, `sem_init()`, `sem_trywait()`, `sem_wait()`
sem_timedwait() Wait on a semaphore, with a timeout

Synopsis:
#include <semaphore.h>
#include <time.h>

int sem_timedwait(
    sem_t * sem,
    const struct timespec * abs_timeout);

Arguments:
sem The semaphore that you want to wait on.

abs_timeout A pointer to a timespec structure that specifies the maximum time to wait to lock the semaphore, expressed as an absolute time.

Library:
libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The sem_timedwait() function locks the semaphore referenced by sem as in the sem_wait() function. However, if the semaphore can’t be locked without waiting for another process or thread to unlock the semaphore by calling sem_post(), the wait is terminated when the specified timeout expires.

The timeout expires when the absolute time specified by abs_timeout passes, as measured by the clock on which timeouts are based (i.e. when the value of that clock equals or exceeds abs_timeout), or if the absolute time specified by abs_timeout has already been passed at the time of the call. The timeout is based on the CLOCK_REALTIME clock.
Returns:

0  The calling process successfully performed the semaphore lock operation on the semaphore designated by sem.

-1  The call was unsuccessful (errno is set). The state of the semaphore is unchanged.

Errors:

EDEADLK  A deadlock condition was detected.
EINTR    A signal interrupted this function.
EINVAL   Invalid semaphore sem, or the thread would have blocked, and the abs_timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.
ETIMEDOUT  The semaphore couldn’t be locked before the specified timeout expired.

Examples:

```c
#include <stdio.h>
#include <semaphore.h>
#include <time.h>

main(){
    struct timespec tm;
    sem_t sem;
    int i=0;

    sem_init( &sem, 0, 0);

    do {
        clock_gettime(CLOCK_REALTIME, &tm);
        tm.tv_sec += 1;
        i++;
        printf("i=%d\n",i);
        if (i==10) {
            sem_post(&sem);
        }
    } while ( sem_timedwait( &sem, &tm ) == -1 );
```
printf("Semaphore acquired after %d timeouts\n", i);
return;
}

Classification:

POSIX 1003.1 SEM TMO

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

See also:

sem_post(), sem_trywait(), sem_wait(), time(), timespec
sem_trywait()  

Wait on a semaphore, but don’t block

Synopsis:
```c
#include <semaphore.h>

int sem_trywait( sem_t * sem );
```

Arguments:
- `sem`: A pointer to the `sem_t` object for the semaphore that you want to wait on.

Library:
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `sem_trywait()` function decrements the semaphore if the semaphore’s value is greater than zero; otherwise, the function simply returns.

Returns:
- `0`: The semaphore was successfully decremented.
- `-1`: The state of the semaphore is unchanged (`errno` is set).

Errors:
- **EAGAIN**: The semaphore was already locked, so it couldn’t be immediately locked by the `sem_trywait()` function.
- **EDEADLK**: A deadlock condition was detected.
- **EINVAL**: Invalid semaphore descriptor `sem`.
- **EINTR**: A signal interrupted this function.
Classification:

POSIX 1003.1 SEM

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sem_destroy(), sem_init(), sem_post(), sem_wait()
sem_unlink()  Destroy a named semaphore

Synopsis:

```
#include <semaphore.h>

int sem_unlink(const char *sem_name);
```

Arguments:

`sem_name` The name of the semaphore that you want to destroy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_unlink()` function destroys the named semaphore, `sem_name`. Open semaphores are removed the same way that `unlink()` removes open files; the processes that have the semaphore open can still use it, but the semaphore will disappear as soon as the last process uses `sem_close()` to close it.

Any attempt to use `sem_open()` on an unlinked semaphore will refer to a new semaphore.

Semaphores are persistent as long as the system remains up.

Returns:

0 Success.

-1 An error occurred (`errno` is set).

Errors:

EACCESS You don’t have permission to unlink the semaphore.

ELOOP Too many levels of symbolic links or prefixes.
sem_unlink()

ENOENT    The semaphore \texttt{sem\_name} doesn’t exist.
ENAMETOOLONG
    The \texttt{sem\_name} argument is longer than \texttt{(NAME\_MAX - 8)}.
ENOSYS    The \texttt{sem\_unlink()} function isn’t implemented for the filesystem specified in \texttt{path}.

Classification:

POSIX 1003.1 SEM

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

\texttt{sem\_open()}, \texttt{sem\_close()}, \texttt{sem\_wait()}, \texttt{sem\_trywait()}, \texttt{sem\_post()}

\texttt{procnto*} in the \texttt{Utilities Reference}
sem_wait() — Wait on a semaphore

Synopsis:

```c
#include <semaphore.h>

int sem_wait( sem_t * sem );
```

Arguments:

- `sem`: A pointer to the `sem_t` object for the semaphore that you want to wait on.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sem_wait()` function decrements the semaphore referred to by the `sem` argument. If the semaphore value is not greater than zero, then the calling process blocks until it can decrement the counter, or the call is interrupted by signal.

Some process should eventually call `sem_post()` to increment the semaphore.

Returns:

- **0**: The semaphore was successfully decremented.
- **-1**: The state of the semaphore is unchanged (`errno` is set).

Errors:

- **EDEADLK**: A deadlock condition was detected.
- **EINVAL**: Invalid semaphore descriptor `sem`.
- **EINTR**: A signal interrupted this function.
Classification:

POSIX 1003.1 SEM

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`sem_destroy()`, `sem_init()`, `sem_post()`, `sem_trywait()`
Send a message to a connected socket

Synopsis:
```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send( int s,
              const void * msg,
              size_t len,
              int flags);
```

Arguments:
- `s` The descriptor for the socket; see `socket()`.
- `msg` A pointer to the message that you want to send.
- `len` The length of the message.
- `flags` A combination of the following:
  - `MSG_OOB` — process out-of-band data. Use this bit when you send “out-of-band” data on sockets that support this notion (e.g. SOCK_STREAM). The underlying protocol must also support out-of-band data.
  - `MSG_DONTROUTE` — bypass routing; create a direct interface. You normally use this bit only in diagnostic or routing programs.

The tiny TCP/IP stack doesn’t support `MSG_OOB` and `MSG_DONTROUTE`. For more information, see `npm-ttcpip.so` in the Utilities Reference.

Library:
```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.
Description:

The `send()`, `sendto()`, and `sendmsg()` functions are used to transmit a message to another socket. The `send()` function can be used only when the socket is in a connected state, while `sendto()` and `sendmsg()` can be used at any time.

The length of the message is given by `len`. If the message is too long to pass atomically through the underlying protocol, the error `EMSGSIZE` is returned, and the message isn’t transmitted.

No indication of failure to deliver is implicit in a `send()`. Locally detected errors are indicated by a return value of `-1`.

If no message space is available at the socket to hold the message to be transmitted, then `send()` normally blocks, unless the socket has been placed in nonblocking I/O mode. You can use `select()` to determine when it’s possible to send more data.

Returns:

The number of bytes sent, or `-1` if an error occurs (`errno` is set).

Errors:

- **EBADF** An invalid descriptor was specified.
- **EDESTADDRREQ** A destination address is required.
- **EFAULT** An invalid user space address was specified for a parameter.
- **EMSGSIZE** The socket requires that the message be sent atomically, but the size of the message made this impossible.
- **ENOBUFS** The system couldn’t allocate an internal buffer. The operation may succeed when buffers become available.
- **ENOTSOCK** The argument `s` isn’t a socket.
send()  

EPIPE  An attempt was made to write to a pipe (or FIFO) that isn’t open for reading by any process. A SIGPIPE signal is also sent to the process.

EW OULD BLOCK  
The socket is marked nonblocking and the requested operation would block.

Classification:  

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:  

getsockopt(), ioctl(), recv(), select(), sendmsg(), sendto(), socket(), write()
Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendmsg( int s,
    const struct msghdr * msg,
    int flags );
```

Arguments:

- **s**: The descriptor for the socket; see `socket()`.
- **msg**: A pointer to the message that you want to send. For a description of the `msghdr` structure, see `recvmsg()`.
- **flags**: A combination of the following:
  - MSG_OOB — process out-of-band data. Use this bit when you send “out-of-band” data on sockets that support this notion (e.g. SOCK_STREAM). The underlying protocol must also support out-of-band data.
  - MSG_DONTROUTE — bypass routing; create a direct interface. You normally use this bit only in diagnostic or routing programs.

The tiny TCP/IP stack doesn’t support MSG_OOB and MSG_DONTROUTE. For more information, see `npm-ttcpip.so` in the Utilities Reference.

Library:

```
libsocket
```

Use the `-l socket` option to `qcc` to link against this library.
Description:

The sendmsg() function is used to transmit a message to another socket. You can use send() only when the socket is in a connected state; you can use sendmsg() at any time.

No indication of failure to deliver is implicit in a sendmsg(). Locally detected errors are indicated by a return value of -1.

If no message space is available at the socket to hold the message to be transmitted, then sendmsg() normally blocks, unless the socket has been placed in nonblocking I/O mode. You can use select() to determine when it’s possible to send more data.

Returns:

The number of bytes sent, or -1 if an error occurs (errno is set).

Errors:

EBADF An invalid descriptor was specified.

EDESTADDRREQ A destination address is required.

EFAULT An invalid user space address was specified for a parameter.

EMSGSIZE The socket requires that the message be sent atomically, but the size of the message made this impossible.

ENOBUFS The system couldn’t allocate an internal buffer. The operation may succeed when buffers become available.

ENOTSOCK The argument s isn’t a socket.

EWOULDBLOCK The socket is marked nonblocking and the requested operation would block.
sendmsg()

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

getsockopt(), ioctl(), recv(), select(), send(), sendto(), socket(), write()
Send a message to a socket at a specific address

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto( int s,
    const void *msg,
    size_t len,
    int flags,
    const struct sockaddr *to,
    socklen_t tolen );
```

Arguments:

- `s` The descriptor for the socket; see `socket()`.
- `msg` A pointer to the message that you want to send.
- `len` The length of the message.
- `flags` A combination of the following:
  - MSG_OOB — process out-of-band data. Use this bit when you send “out-of-band” data on sockets that support this notion (e.g. SOCK_STREAM). The underlying protocol must also support out-of-band data.
  - MSG_DONTROUTE — bypass routing; create a direct interface. You normally use this bit only in diagnostic or routing programs.

The tiny TCP/IP stack doesn’t support MSG_OOB and MSG_DONTROUTE. For more information, see `npm-ttcpip.so` in the Utilities Reference.

- `to` A pointer to a `sockaddr` object that specifies the address of the target.
- `tolen` A `socklen_t` object that specifies the size of the `to` address.
Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The sendto() function is used to transmit a message to another socket. You can use send() only when the socket is in a connected state; you can use sendto() at any time.

The address of the target is given by to, with tolen specifying its size. The length of the message is given by len. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message isn’t transmitted.

No indication of failure to deliver is implicit in a sendto(). Locally detected errors are indicated by a return value of -1.

If no message space is available at the socket to hold the message to be transmitted, then sendto() normally blocks, unless the socket has been placed in nonblocking I/O mode. You can use select() to determine when it’s possible to send more data.

Returns:

The number of bytes sent, or -1 if an error occurs (errno is set).

Errors:

EBADF An invalid descriptor was specified.

EDESTADDRREQ A destination address is required.

EFAULT An invalid user space address was specified for a parameter.

EMSGSIZE The socket requires that the message be sent atomically, but the size of the message made this impossible.
sendto()

ENOBUFFS The system couldn’t allocate an internal buffer. The operation may succeed when buffers become available.

ENOTSOCK The argument s isn’t a socket.

EWOULDBLOCK The socket is marked nonblocking and the requested operation would block.

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getsockopt(), ioctl(), recv(), select(), send(), sendmsg(), socket(), write()
**Synopsis:**

```c
#include <netdb.h>

struct servent {
    char * s_name;
    char ** s_aliases;
    int s_port;
    char * s_proto;
};
```

**Description:**

This structure is used to hold the broken-out fields of a line in the network services database, `/etc/services`. The members of this structure are:

- **s_name**: The name of the service.
- **s_aliases**: A zero-terminated list of alternate names for the service.
- **s_port**: The port number that the service resides at. Port numbers are returned in network byte order.
- **s_proto**: The name of the protocol to use when contacting the service.

**Classification:**

POSIX 1003.1

**See also:**

- `endservent()`, `getservbyname()`, `getservbyport()`, `getservent()`, `setservent()`
- `/etc/services` in the Utilities Reference
Associate a buffer with a stream

**Synopsis:**

```c
#include <stdio.h>

void setbuf( FILE *fp,
             char *buffer );
```

**Arguments:**

- `fp` The stream that you want to associate with a buffer.
- `buffer` NULL, or a pointer to the buffer; see below.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setbuf()` function associates the supplied `buffer` with the stream specified by `fp`. If you want to call `setbuf()`, you must call it after opening the stream, but before doing any reading, writing, or seeking.

If `buffer` is NULL, all input/output for the stream is completely unbuffered. If `buffer` isn’t NULL, then it must point to an array that’s at least `BUFSIZ` characters long, and all input/output is fully buffered.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char *buffer;
    FILE *fp;

    buffer = (char *)malloc( BUFSIZ );
    if( buffer == NULL ) {
        return EXIT_FAILURE;
    }
```


```c
fp = fopen( "some_file", "r" );
setbuf( fp, buffer );
/* . */
/* . */
/* . */
fclose( fp );
free( buffer );
return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`fopen()`, `setvbuf()`
Assign block buffering to a stream

Synopsis:

```c
#include <unix.h>

void setbuffer( FILE *iop,
                char *abuf,
                size_t asize );
```

Arguments:

- `iop` The stream that you want to set the buffering for.
- `abuf` NULL, or a pointer to the buffer that you want the stream to use.
- `asize` The size of the buffer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setbuffer()` and `setlinebuf()` functions assign buffering to a stream. The types of buffering available are:

- **Unbuffered**: Information appears on the destination file or terminal as soon as written.
- **Block-buffered**: Many characters are saved and written as a block.
- **Line-buffered**: Characters are saved until either a newline is encountered or input is read from `stdin`.

You can use `fflush()` to force the block out early. Normally all files are block-buffered. A buffer is obtained from `malloc()` when you perform the first `getc()` or `putc()` on the file. If the standard stream `stdout` refers
to a terminal, it’s line-buffered. The standard stream stderr is unbuffered by default.
If you want to use setbuffer(), you must call it after opening the stream, but before doing any reading or writing. It uses the character array abuf, whose size is given by asize, instead of an automatically allocated buffer. If abuf is NULL, input and output are completely unbuffered. A manifest constant BUFSIZ, defined in the <stdio.h> header, tells how large an array is needed:

```c
char buf[BUFSIZ];
```

You can use freopen() to change a stream from unbuffered or line-buffered to block buffered. To change a stream from block-buffered or line-buffered to unbuffered, call freopen(), and then call setbuf() with a buffer argument of NULL.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.
See also:

fclose(), fflush(), fopen(), fread(), freopen(), getc(), malloc(), printf(), putc(), puts(), setbuf(), setlinebuf(), setvbuf()
setdomainname()

Set the domain name of the current host

Synopsis:

```c
#include <unistd.h>

int setdomainname( const char * name,
                   size_t namelen );
```

Arguments:

- `name` The domain name.
- `namelen` The length of the name.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `setdomainname()` function sets the domain `name` of the host machine. Only the superuser (`root`) can use this function and even then, the function is normally used only when bootstrapping a system.

Returns:

- `0` Success.
- `-1` Failure; `errno` is set.

Errors:

- `EFAULT` The `name` or `namelen` parameters gave an invalid address.
- `EPERM` The caller tried to set the domain name without being the superuser.
setdomainname()

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getdomainname()
Synopsis:

```c
#include <unistd.h>

int setegid( gid_t gid );
```

Arguments:

- `gid` The effective group ID that you want to use for the process.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setegid()` function lets the calling process set the effective group ID based on the following:

- If the process is the superuser, the `setegid()` function sets the effective group ID to `gid`.
- If the process isn’t the superuser, but `gid` is equal to the real group ID or saved set-group ID, `setegid()` sets the effective group ID to `gid`.

The real and saved group ID aren’t changed.

If a set-group ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group ID.

The “superuser” is defined as any process with an effective user ID of 0, or an effective user ID of `root`.

**setegid()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Returns:**

Zero for success, or -1 if an error occurs (errno is set).

**Errors:**

EINVAl The value of gid is out of range.

EPERM The process isn’t the superuser, and gid doesn’t match the real group ID or the saved set-group ID.

**Examples:**

```c
/*
 * This process sets its effective group ID to 2
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main( void )
{
    gid_t oegid;

    oegid = getegid();
    if( setegid( 2 ) == -1 ) {
        perror( "setegid" );
        return EXIT_FAILURE;
    }

    printf( "Was effective group %d, is 2\n", oegid );
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued…
setegid()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno`, `getegid()`, `seteuid()`, `setgid()`, `setuid()`
Setenv()

Create or change an environment variable

Synopsis:

```
#include <stdlib.h>

int setenv( const char* name,
            const char* value,
            int overwrite );
```

Arguments:

- `name` The name of the environment variable that you want to set.
- `value` NULL, or the value for the environment variable; see below.
- `overwrite` A nonzero value if you want the function to overwrite the variable if it exists, or 0 if you don’t want to overwrite the variable.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setenv()` function sets the environment variable `name` to `value`. If `name` doesn’t exist in the environment, it’s created; if `name` exists and `overwrite` is nonzero, the variable’s old value is overwritten with `value`; otherwise, it isn’t changed.

! **CAUTION:** This function doesn’t free any memory. If you want to change the value of an existing environment variable, you should use `putenv()` instead.
Copies of the specified name and value are placed in the environment. If value is NULL, the environment variable specified by name is removed from the environment.

The value of the global environ pointer could be changed by a call to the setenv() function.

Environment variable names are case-sensitive.

**Returns:**

0 Success.

Nonzero An error occurred (errno is set).

**Errors:**

ENOMEM Not enough memory to allocate a new environment variable.

**Examples:**

Change the string assigned to INCLUDE and then display the new string:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char* path;

    if( setenv( "INCLUDE",
                "/usr/nto/include:/home/fred/include",
                1 ) == 0 ) {
        if( (path = getenv( "INCLUDE" )) != NULL ) {
            printf( "INCLUDE=%s\n", path );
        }
    }

    return EXIT_SUCCESS;
}
```

September 10, 2007
setenv()

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

The `setenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

clearenv(), errno, execl(), execle(), execvp(), execve(), execvpe(), getenv(), putenv(), searchenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system(), unsetenv()
**seteuid()**

Set the effective user ID

**Synopsis:**

```c
#include <unistd.h>

int seteuid( uid_t uid );
```

**Arguments:**

- `uid` The effective user ID that you want to use for the process.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `seteuid()` function lets the calling process set the effective user ID, based on the following:

- If the process is the superuser, the `seteuid()` function sets the effective user ID to `uid`.

- If the process isn’t the superuser, and `uid` is equal to the real user ID or saved set-user ID, `seteuid()` sets the effective user ID to `uid`.

The real and saved user IDs aren’t changed.

---

If a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-UID.

The “superuser” is defined as any process with an effective user ID of 0, or an effective user ID of `root`.

**Returns:**

- 0 Success.
- -1 An error occurred (`errno` is set).
Errors:

EINV AL The value of \textit{uid} is out of range.

EPERM The process isn’t the superuser, and \textit{uid} doesn’t match the real user ID or the saved set-user ID.

Examples:

\begin{verbatim}
/*
 * This process sets its effective userid to 0 (root).
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    uid_t oeuid;

    oeuid = geteuid();
    if( seteuid( 0 ) == -1 ) {
        perror( "seteuid" );
        return EXIT_FAILURE;
    }

    printf( "effective userid now 0, was \%d\n", oeuid );
    return EXIT_SUCCESS;
}
\end{verbatim}

Classification:

POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes

\textit{continued…}
seteuid()

Safety

Thread  Yes

See also:

*errno, geteuid(), setegid(), setuid(), setgid()*
**setgid()**

Set the real, effective and saved group IDs

**Synopsis:**

```c
#include <unistd.h>

int setgid( gid_t gid );
```

**Arguments:**

- `gid` The group ID that you want to use for the process.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setgid()` function lets the calling process set the real, effective and saved group IDs, based on the following:

- If the process is the superuser, the `setgid()` function sets the real group ID, effective group ID and saved group ID to `gid`.

- If the process isn’t the superuser, but `gid` is equal to the real group ID, `setgid()` sets the effective group ID to `gid`; the real and saved group IDs aren’t changed.

This function doesn’t change any supplementary group IDs of the calling process.

If you wish to change only the effective group ID, and even if you are the superuser, you should consider using the `setegid()` function.

The “superuser” is defined as any process with an effective user ID of 0, or an effective user ID of `root`. 
**setgid()**

**Returns:**

0    Success.

-1   An error occurred; *errno* is set to indicate the error.

**Errors:**

EINV   The value of *gid* is invalid.

EPERM  The process doesn’t have appropriate privileges, and *gid* doesn’t match the real group ID.

**Examples:**

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    gid_t ogid;

    ogid = getgid();
    if( setgid( 2 ) == -1 ) {
        perror( "setgid" );
        return EXIT_FAILURE;
    }
    printf( "group id is now 2, was %d\n", ogid );
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

*continued…*
setgid()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, setgid(), seteuid(), setuid()
setgrent()

Rewind to the start of the group database file

Synopsis:

```c
#include <grp.h>

int setgrent( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setgrent()` function rewinds to the start of the group name database file. It’s provided for programs that make multiple lookups in the group database (using the `getgrgid()` and `getgrnam()` calls) to avoid the default opening and closing of the group database for each access.

Returns:

- `0` Success.
- `-1` An error occurred.

Errors:

The `setgrent()` function uses `fopen()`. As a result, `errno` can be set to an error for the `fopen()` call.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>continued…</td>
</tr>
</tbody>
</table>

September 10, 2007
**setgrent()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`endgrent()`, `getgrent()`
setgroups()  
Set supplementary group IDs

Synopsis:

```c
#include <unistd.h>

int setgroups( int ngroups,
               const gid_t *gidset );
```

Arguments:

- `ngroups`  The number of entries in the `gidset` array.
- `gidset`  An array of the supplementary group IDs that you want to assign to the current user. This number of entries in this array can’t exceed `NGROUPS_MAX`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setgroups()` function sets the group access list of the current user to the array of group IDs in `gidset`.

- Only `root` can set new groups.

Returns:

- 0, or -1 if an error occurred (`errno` is set).

Errors:

- `EFAULT`  The `gidset` argument isn’t a valid pointer.
- `EPERM`  The caller isn’t `root`.  

September 10, 2007
setgroups()

Classification:
Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

getgroups(), initgroups()
sethostent()
Open the host database file

Synopsis:

#include <netdb.h>

void sethostent( int stayopen );

Arguments:

stayopen Nonzero if you want all queries to the name server to use TCP and you want the connection to be retained after each call to gethostbyname() or gethostbyaddr(). If the stayopen flag is zero, queries use UDP datagrams.

Library:

libsocket

Use the -l socket option to qcc to link against this library.

Description:

The sethostent() routine opens the host database file.

You can use the sethostent() function to request the use of a connected TCP socket for queries.

Classification:

POSIX 1003.1

Safety

Cancellation point No
Interrupt handler No
Signal handler No
Thread No
Caveats:

This function uses static data; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

See also:

endhostent(), gethostbyaddr(), gethostbyname(), gethostent(),
gethostent_r(), hostent

/etc/hosts, /etc/resolv.conf in the Utilities Reference
sethostname()
Set the name of the current host

Synopsis:
#include <unistd.h>

int sethostname( const char * name,
                 size_t namelen );

Arguments:
name The name that you want to use for the host machine.
Hostnames are limited to MAXHOSTNAMELEN characters (defined in <sys/param.h>).

namelen The length of the name.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The sethostname() function sets the name of the host machine to be name. Only the superuser can call this function; this is normally done only at boot time.

This function sets the value of the _CS_HOSTNAME configuration string, not that of the HOSTNAME environment variable.

Returns:
0 Success.
-1 An error occurred (errno is set).
**sethostname()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Errors:**

- **EFAULT** Either *name* or *namelen* gave an invalid address.
- **EPERM** Although the caller wasn’t the superuser, it tried to set the hostname.

**Classification:**

Unix

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

This function is restricted to the superuser, and is normally used only at boot time.

**See also:**

`gethostname()`
SETIOV()

Fill in the fields of an iov_t structure

Synopsis:

```c
#include <unistd.h>

void SETIOV( iov_t *msg,
              void *addr,
              size_t len );
```

Arguments:

- `msg` A pointer to the iov_t structure that you want to set.
- `addr` The value you want to use for the structure’s iov_base member.
- `len` The value you want to use for the structure’s iov_len member.

Description:

The `SETIOV()` macro fills in the fields of an iov_t message structure. The iov_t structure consists of two fields:

```c
typedef struct iovec {
    void   *iov_base;
    size_t  iov_len;
} iov_t;
```

`SETIOV()` doesn’t make a copy of the data that `addr` points to; it just copies the pointer.

Classification:

QNX Neutrino

Safety

Cancellation point  No

continued...


**SETIOV()**

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`GETIOVBASE()`, `GETIOVLLEN()`, `MsgKeyData()`, `MsgReadv()`, `MsgReceivev()`, `MsgReplyv()`, `MsgSendv()`, `MsgWritev()`
setitimer()

Set the value of an interval timer

Synopsis:

```c
#include <sys/time.h>

int setitimer ( int which,
    const struct itimerval *value,
    struct itimerval *ovalue );
```

Arguments:

- `which` The interval time whose value you want to set. Currently, this must be ITIMER_REAL.
- `value` A pointer to a `itimerval` structure that specifies the value that you want to set the interval timer to.
- `ovalue` NULL, or a pointer to a `itimerval` structure where the function can store the old value of the interval timer.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The system provides each process with interval timers, defined in `<sys/time.h>`. The `setitimer()` function sets the value of the timer specified by `which` to the value specified in the structure pointed to by `value`, and if `ovalue` isn’t NULL, stores the previous value of the timer in the structure it points to.

A timer value is defined by the `itimerval` structure (see `gettimeofday()` for the definition of `timeval`), which includes the following members:

```c
struct timeval it_interval; /* timer interval */
struct timeval it_value; /* current value */
```
The `it_value` member indicates the time to the next timer expiration. The `it_interval` member specifies a value to be used in reloading `it_value` when the timer expires. Setting `it_value` to 0 disables a timer, regardless of the value of `it_interval`. Setting `it_interval` to 0 disables a timer after its next expiration (assuming `it_value` is nonzero).

Time values smaller than the resolution of the system clock are rounded up to the resolution of the system clock.

The only supported timer is `ITIMER_REAL`, which decrements in real time. A `SIGALRM` signal is delivered when this timer expires.

The `SIGALRM` so generated isn’t maskable on this bound thread by any signal-masking function, `pthread_sigmask()`, or `sigprocmask()`.

**Returns:**

0  Success.

-1  An error occurred; `errno` is set.

**Errors:**

`EINVAL`  The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the `which` argument is unrecognized.

**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
Caveats:

All flags to `setitimer()` other than ITIMER_REAL behave as documented only with “bound” threads. Their ability to mask the signal works only with bound threads. If the call is made using one of these flags from an unbound thread, the system call returns -1 and sets `errno` to EACCES.

These behaviors are the same for bound or unbound POSIX threads. A POSIX thread with system-wide scope, created by the call:

```c
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

is equivalent to a Solaris bound thread. A POSIX thread with local process scope, created by the call:

```c
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

is equivalent to a Solaris unbound thread.

The microseconds field shouldn’t be equal to or greater than one second.

The `setitimer()` function is independent of `alarm()`.

Don’t use `setitimer(ITIMER_REAL)` with the `sleep()` routine. A `sleep()` call wipes out knowledge of the user signal handler for SIGALRM.

The granularity of the resolution of the alarm time is platform-dependent.

See also:

`alarm()`, `getitimer()`, `gettimeofday()`, `pthread_attr_setscope()`, `pthread_sigmask()`, `sigprocmask()`, `sleep()`, `sysconf()`
**setjmp()**

Save the calling environment for **longjmp()**

**Synopsis:**

```c
#include <setjmp.h>

int setjmp( jmp_buf env );
```

**Arguments:**

- `env` A buffer where the function can save the calling environment.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setjmp()` function saves the calling environment in its `env` argument for use by the `longjmp()` function.

Error handling can be implemented by using `setjmp()` to record the point to return to following an error. When an error is detected in a function, that function uses `longjmp()` to jump back to the recorded position. The original function that called `setjmp()` must still be active (that is, it can’t have returned to the function that called it).

Be careful to ensure that any resources (allocated memory, opened files, etc) are cleaned up properly.

---

**WARNING:** Do not use `longjmp()` or `siglongjmp()` to restore an environment saved by a call to `setjmp()` or `sigsetjmp()` in another thread. If you’re lucky, your application will crash; if not, it’ll look as if it works for a while, until random scribbling on the stack causes it to crash.
Return:

Zero on the first call, or nonzero if the return is the result of a call to the `longjmp()` function.

Examples:

```c
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf env;

void rtn( void )
{
    printf( "about to longjmp()\n" );
    longjmp( env, 14 );
}

int main( void )
{
    int ret_val;

    ret_val = setjmp( env );

    if( ret_val == 0 ) {
        printf( "after setjmp(): %d\n", ret_val );
        rtn();
        printf( "back from rtn(): %d\n", ret_val );
    } else {
        printf( "back from longjmp(): %d\n", ret_val );
    }

    return EXIT_SUCCESS;
}
```

produces the output:

```
after setjmp(): 0
about to longjmp()
back from longjmp(): 14
```
setjmp()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

longjmp()
setkey()

Set the key used in encryption

Synopsis:

```c
#include <stdlib.h>

void setkey( const char *__key );
```

Arguments:

```c
__key    A 64-character array of binary values (numeric 0 or 1).
```

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `setkey()` function allows limited access to the NBS Data Encryption Standard (DES) algorithm itself. It derives a 56-bit key from the given `__key` by dividing the array into groups of 8 and ignoring the last bit in each group.

Classification:

POSIX 1003.1 XSI

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
setkey()

See also:

crypt(), encrypt()
setlinebuf()
Assign line buffering to a stream

Synopsis:
#include <unix.h>

int setlinebuf( FILE *iop );

Arguments:

iop The stream that you want to use line buffering.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The setbuffer() and setlinebuf() functions assign buffering to a stream. The types of buffering available are:

Unbuffered Information appears on the destination file or terminal as soon as written.
Block-buffered Many characters are saved and written as a block.
Line-buffered Characters are saved until either a newline is encountered or input is read from stdin.

You can use fflush() to force the block out early. Normally all files are block-buffered. A buffer is obtained from malloc() when the first getc() or putc() is performed on the file. If the standard stream stdout refers to a terminal, it's line-buffered. The standard stream stderr is unbuffered by default.

You can use setlinebuf() to change the buffering on a stream from block-buffered or unbuffered to line-buffered. Unlike setbuffer(), you can call setlinebuf() at any time that the stream iop is active.
You can use `freopen()` to change a stream from unbuffered or line-buffered to block buffered. To change a stream from block-buffered or line-buffered to unbuffered, call `freopen()`, and then call `setbuf()` with a buffer argument of NULL.

**Returns:**

No useful value.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`fclose()`, `fflush()`, `fopen()`, `fread()`, `freopen()`, `getc()`, `malloc()`, `printf()`, `putc()`, `puts()`, `setbuf()`, `setbuffer()`, `setvbuf()`
setlocale()

Select a program’s locale

Synopsis:

```c
#include <locale.h>

char * setlocale( int category,
                  const char * locale );
```

Arguments:

- `category`: The part of the environment that you want to set; one of:
  - `LC_ALL` — select the entire locale environment.
  - `LC_COLLATE` — select only the collating sequence.
  - `LC_CTYPE` — select only the character-handling information.
  - `LC_MESSAGES` — specify the language to be used for messages.
  - `LC_MONETARY` — select only monetary formatting information.
  - `LC_NUMERIC` — select only the numeric-format environment.
  - `LC_TIME` — select only the time-related environment.

- `locale`: The locale that you want to use. The following built-in locales are offered:
  - `C` (default)
  - `C-TRADITIONAL`
  - `POSIX`

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**Description:**

The `setlocale()` function selects a program’s locale, according to the specified category and the specified locale.

A locale affects several things:

- The collating sequence (the order in which characters compare with one another) used by `strcoll()` or `wcscoll()`.
- The way certain character-handling functions (such as `isalnum()` and `isalpha()`) operate. The wide-character versions include `iswalnum()` and `iswalpha()`.
- The decimal-point character used in formatted input/output and string conversion (`printf()`, `scanf()`, and friends).
- The format and names used in the string produced by the `strftime()` and `wcsftime()` functions.

See the `localeconv()` function for more information about the locale.

At the start of a program, the default C locale is initialized as if the following call to `setlocale()` appeared at the start of `main()`:

```c
(void) setlocale( LC_ALL, "C" );
```

**Returns:**

The string associated with the specified category for the new locale, or NULL if an error occurs. This function doesn’t change the program’s locale when error occurs.

For a NULL pointer of locale, this function returns a pointer to the string associated with the category for the program’s current locale. This function doesn’t change the program’s locale.
setlocale()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalpha(), isascii(), localeconv(), printf(), scanf(), strcat(), strchr(), strcmp(), strcoll(), strcpy(), strcat(), strftime(), strlen(), strpbrk(), strspn(), strtol(), strtok(), strxfrm(), tm
setlogmask()

Set the system log priority mask

Synopsis:

```c
#include <syslog.h>

int setlogmask( int maskpri );
```

Arguments:

- `maskpri` The new log priority mask; see below.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setlogmask()` function sets the log priority mask to `maskpri` and returns the previous mask. Calls to `syslog()` or `vsyslog()` with a priority that isn’t set in `maskpri` are rejected.

You can calculate the mask for an individual priority `pri` with the macro:

```c
LOG_MASK(pri);
```

You can get the mask for all priorities up to and including `toppri` with the macro:

```c
LOG_UPTO(toppri);
```

The default allows all priorities to be logged. See the `syslog()` function for a list of the priorities.

Returns:

The previous log mask level.
Examples:

See syslog().

Classification:

POSIX 1003.1 XSI

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

closelog(), openlog(), syslog(), vsyslog()

logger, syslogd in the Utilities Reference
**Synopsis:**

```c
#include <netdb.h>

void setnetent( int stayopen );
```

**Arguments:**

`stayopen`  Nonzero if you don’t want the network database to be closed after each call to `getnetbyname()` or `getnetbyaddr()`.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `setnetent()` function opens and rewinds the network name database file.

**Files:**

`/etc/networks`

Network name database file.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
Caveats:

This function uses static data; if you need the data for future use, you should copy it before any subsequent calls overwrite it.

See also:

endnetent(), getnetbyaddr(), getnetbyname(), getnetent(), netent

/etc/networks in the Utilities Reference
**setpgid()**

Join or create a process group

**Synopsis:**

```c
#include <process.h>

int setpgid( pid_t pid,
             pid_t pgid );
```

**Arguments:**

- **pid**: 0, or the ID of the process whose process group you want to set.
- **pgid**: 0, or the process group ID that you want to join or create.

**Library:**

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setpgid()` function is used either to join an existing process group or to create a new process group within the session of the calling process. The process group ID of a session leader doesn’t change.

The following definitions are worth mentioning:

- **Process**: An executing instance of a program, identified by a nonnegative integer called a process ID.
- **Process group**: A collection of one or more processes, with a unique process group ID. A process group ID is a positive integer.

On successful completion, the process group ID of the process with a process ID matching `pid` is set to `pgid`. As a special case, you can specify either `pid` or `pgid` as zero, meaning that the process ID of the calling process is to be used.
Returns:

0    Success.

-1    An error occurred; errno is set.

Errors:

EACCES    The value of the pid argument matches the process ID of a child process of the calling process, and the child process has successfully executed one of the exec*( ) functions.

EINVAL    The value of pgid is less than zero.

ENOSYS    The setpgid() function isn’t supported by this implementation (included for POSIX compatibility).

EPERM    The calling process doesn’t have sufficient privilege to set the process group id pgid on process pid.

ESRCH    The process pid doesn’t exist.

Examples:

/*
 * The process joins process group 0.
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <process.h>

int main( void )
{
    if( setpgid( getpid(), 0 ) == -1 ) {
        perror( "setpgid" );
    }
    printf( "%d belongs to process group %d\n",
            getpid(), getpgrp() );
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, getpgid(), getsid(), setsid()
setpgrp()
Set the process group

Synopsis:
#include <unistd.h>

pid_t setpgrp( void );

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
If the calling process isn’t already a session leader, setpgrp() makes it one by setting its process group ID and session ID to the value of its process ID, and releases its controlling terminal.

Returns:
The new process group ID.

Classification:
POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
setpgrp()

See also:

`execl()`, `execle()`, `exclp()`, `execle()`, `execv()`, `execve()`, `execvp()`,  
`execvpe()`, `fork()`, `getpid()`, `getpgrp()`, `getsid()`, `kill()`, `signal()`
**setprio()**

*Set the priority of a process*

**Synopsis:**

```c
#include <sched.h>

int setprio( pid_t pid,
             int prio );
```

**Arguments:**

- `pid` The process ID of the process whose priority you want to set.
- `prio` The new priority.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setprio()` function changes the priority of thread 1 of process `pid` to priority `prio`. If `pid` is zero, the priority of the calling thread is set.

**Returns:**

The previous priority, or -1 if an error occurred (`errno` is set).

**Errors:**

- `EINVAL` The priority `prio` isn’t a valid priority.
- `EPERM` The calling process doesn’t have sufficient privilege to set the priority.
- `ESRCH` The process `pid` doesn’t exist.
Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The `getprio()` and `setprio()` functions are included in the QNX Neutrino libraries for porting QNX 4 applications. For new programs, use `sched_setparam()` or `pthread_setschedparam()`.

See also:

`errno`, `getprio()`, `pthread_getschedparam()`,
`pthread_setschedparam()`, `sched_getparam()`,
`sched_get_priority_max()`, `sched_get_priority_min()`,
`sched_getscheduler()`, `sched_setscheduler()`, `sched_yield()`
**setprotoent()**

*Open the protocol name database file*

**Synopsis:**

```c
#include <netdb.h>

void setprotoent( int stayopen );
```

**Arguments:**

- `stayopen` Nonzero if you don’t want the database to be closed after each call to `getprotobyname()` or `getprotobynumber()`.

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `setprotoent()` function opens and rewinds the protocol name database file. If the `stayopen` flag is nonzero, the protocol name database isn’t closed after each call to `getprotobyname()` or `getprotobynumber()`.

**Files:**

`/etc/protocols`

Protocol name database file.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

*continued...*
setprotoent()

Safety

<table>
<thead>
<tr>
<th>Signal handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

This function uses static data; if you need the data for future use, you should copy it before any subsequent calls overwrite it.

Currently, only the Internet protocols are understood.

See also:

endprotoent(), getprotobynumber(), getprotoent(), protoent

/etc/protocols in the Utilities Reference
setpwent()
Rewind the password database file

Synopsis:

```c
#include <sys/types.h>
#include <pwd.h>

int setpwent( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setpwent()` function rewinds to the start of the password name database file. It’s provided for programs that make multiple lookups in the password database (using the `getpwuid()` and `getpwnam()` calls) to avoid opening and closing the password database for each access.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`endpwent()`, `getpwent()`
setregid()  © 2007, QNX Software Systems GmbH & Co. KG.
Set real and effective group IDs

Synopsis:

```c
#include <unistd.h>

int setregid( gid_t rgid,
              gid_t egid );
```

Arguments:

- `rgid` The real group ID that you want to use for the process, or -1 if you don’t want to change it.
- `egid` The effective group ID that you want to use for the process, or -1 if you don’t want to change it.

Library:

```
lIBC
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setregid()` function sets the real and effective group IDs of the calling process. If `rgid` or `egid` is -1, the corresponding real or effective group ID is left unchanged.

If the effective user ID of the calling process is the superuser, you can set the real group ID and the effective group ID to any legal value.

If the effective user ID of the calling process isn’t the superuser, you can set either the real group ID to the saved set-group ID, or the effective group ID to either the saved set-group ID or the real group ID.
If a set-group ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group ID.

The “superuser” is defined as any process with an effective user ID of 0, or an effective user ID of root.

In either case, if you’re changing the real group ID (i.e. rigid isn’t -1), or you’re changing the effective group ID to a value that isn’t equal to the real group ID, the saved set-group ID is set to the new effective group ID.

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EINVAL The rigid or egid is out of range.

EPERM The calling process isn’t the superuser, and you tried to change the effective group ID to a value other than the real or saved set-group ID.

Or:

The calling process isn’t the superuser, and you tried to change the real group ID to a value other than the effective group ID.

Classification:

POSIX 1003.1 XSI
setregid()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

extexec(), getgid(), setreuid(), setuid()
setreuid()
Set real and effect user IDs

Synopsis:

```
#include <unistd.h>

int setreuid( uid_t ruid,
              uid_t euid );
```

Arguments:

- `ruid` The real user ID that you want to use for the process, or -1 if you don’t want to change it.
- `euid` The effective user ID that you want to use for the process, or -1 if you don’t want to change it.

Library:

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `setreuid()` function sets the real and effective user IDs of the calling process. If `ruid` or `euid` is -1, the corresponding real or effective user ID isn’t changed.

If the effective user ID of the calling process is the superuser, you can set the real user ID and the effective user ID to any legal value.

If the effective user ID of the calling process isn’t the superuser, you can set either the real user ID to the effective user ID, or the effective user ID to the saved set-user ID or the real user ID.

If a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-UID.

In either case, if you’re changing the real user ID (i.e. `ruid` is not -1), or you’re changing the effective user ID to a value that isn’t equal to
the real user ID, the saved set-user ID is set equal to the new effective user ID.
The “superuser” is defined as any process with an effective user ID of 0, or an effective user ID of root.

Returns:

Zero on success, or -1 if an error occurs (errno is set).

Errors:

EINV AL The ruid or euid is out of range.
EPERM The calling process isn’t the superuser, and you tried to change the effective user ID to a value other than the real or saved set-user ID.

Or

The calling process isn’t the superuser, and you tried to change the real user ID to a value other than the effective user ID.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
setreuid()

See also:

execve(), getuid(), setregid(), setuid()
**Synopsis:**

```c
#include <sys/resource.h>

int setrlimit( int resource,
               const struct rlimit * rlp);

int setrlimit64( int resource,
                 const struct rlimit64 * rlp);
```

**Arguments:**

- `resource` The resource whose limit you want to get. For a list of the possible resources, their descriptions, and the actions taken when the current limit is exceeded, see below.

- `rlp` A pointer to a `rlimit` or `rlimit64` structure that specifies the limit that you want to set for the resource. The `rlimit` and `rlimit64` structures include at least the following members:

  ```c
  rlim_t rlim_cur; /* current (soft) limit */
  rlim_t rlim_max; /* hard limit */
  ```

  The `rlim_t` type is an arithmetic data type to which you can cast objects of type `int`, `size_t`, and `off_t` without loss of information.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setrlimit()` function sets the limits on the consumption of a variety of system resources by a process and each process it creates. The `setrlimit64()` function is a 64-bit version of `setrlimit()`.
Each call to `setrlimit()` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit.

A process can change soft limits to any value that’s less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that’s greater than or equal to the soft limit. Only a process with an effective user ID of superuser can raise a hard limit. Both hard and soft limits can be changed in a single call to `setrlimit()`, subject to the constraints described above. Limits may have an “infinite” value of RLIM_INFINITY.

RLIM_INFINITY is a special value, and it’s actual numerical value doesn’t represent a valid VM/AS size in bytes.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized below:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_AS</td>
<td>(POSIX) The maximum size, in bytes, of a process’s mapped address space.</td>
<td>If the limit is exceeded, the <code>brk()</code>, <code>mmap()</code> and <code>sbrk()</code> functions fail with <code>errno</code> set to ENOMEM. In addition, the automatic stack growth fails with the effects outlined above.</td>
</tr>
<tr>
<td>RLIMIT_CORE</td>
<td>(POSIX) The maximum size, in bytes, of a core file that a process may create. A limit of 0 prevents the creation of a core file.</td>
<td>The writing of a core file terminates at this size.</td>
</tr>
</tbody>
</table>

continued...
## setrlimit(), setrlimit64()

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_CPU</td>
<td>(POSIX) The maximum amount of CPU time, in seconds, that a process may use. This is a soft limit only.</td>
<td>SIGXCPU is sent to the process. If the process is holding or ignoring SIGXCPU, the behavior is defined by the scheduling class.</td>
</tr>
<tr>
<td>RLIMIT_DATA</td>
<td>(POSIX) The maximum size, in bytes, that a process’s heap may be.</td>
<td>The brk() function fails with errno set to ENOMEM.</td>
</tr>
<tr>
<td>RLIMITFSIZE</td>
<td>(POSIX) The maximum size, in bytes, of a file that a process may create. A limit of 0 prevents the creation of a file.</td>
<td>The SIGXFSZ signal is sent to the process. If the process is holding or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit fail with errno set to EFBIG.</td>
</tr>
<tr>
<td>RLIMIT_MEMLOCK</td>
<td>The maximum amount of memory that can be locked into physical memory (so it will never be paged out).</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_FILENO</td>
<td>(POSIX) One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors that a process may create.</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_NPROC</td>
<td>The maximum number of processes.</td>
<td>Attempts to create new processes fail.</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_NTHR</td>
<td>The maximum number of threads.</td>
<td>Attempts to create threads (e.g. by calling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pthread_create() fail.</td>
</tr>
<tr>
<td>RLIMIT_OFILE</td>
<td>Same as RLIMIT_NOFILE.</td>
<td></td>
</tr>
<tr>
<td>RLIMIT_RSS</td>
<td>The maximum resident set size for a process; the same as RLIMIT_AS.</td>
<td>Same as RLIMIT_AS.</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_STACK</td>
<td>(POSIX) The maximum size, in bytes, that a process’s stack may be. The system will not automatically grow the stack beyond this limit. Within a process, <code>setrlimit()</code> increases the limit on the size of your stack, but doesn’t move current memory segments to allow for that growth. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used. Within a multithreaded process, <code>setrlimit()</code> has no impact on the stack size limit for the calling thread if the calling thread isn’t the main thread. A call to <code>setrlimit()</code> for RLIMIT_STACK impacts only the main thread’s stack, and should be made only from the main thread, if at all.</td>
<td>The SIGSEGV signal is sent to the process. If the process is holding or ignoring SIGSEGV, or is catching SIGSEGV and hasn’t made arrangements to use an alternate stack, the disposition of SIGSEGV is set to SIG_DFL before it’s sent.</td>
</tr>
<tr>
<td>RLIMIT_VMEM</td>
<td>Same as RLIMIT_AS.</td>
<td>Same as RLIMIT_AS.</td>
</tr>
</tbody>
</table>

Because limit information is stored in the per-process information, the shell builtin `ulimit` command (see the entry for `ksh` in the Utilities
setrlimit(), setrlimit64()

Reference) must directly execute this system call if it’s to affect all future processes created by the shell. The values of the current limit of the following resources affect these parameters:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_FSIZE</td>
<td>FCHR_MAX</td>
</tr>
<tr>
<td>RLIMIT_NOFILE</td>
<td>OPEN_MAX</td>
</tr>
</tbody>
</table>

When using the setrlimit() function, if the requested new limit is RLIM_INFINITY, there’s no new limit; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit is the corresponding saved hard limit; otherwise, if the requested new limit is RLIM_SAVED_CUR, the new limit is the corresponding saved soft limit; otherwise, the new limit is the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type rlim_t, then it’s overwritten with the new limit.

The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless a previous call to getrlimit() returned that value as the soft or hard limit for the corresponding resource limit.

A limit whose value is greater than RLIM_INFINITY is permitted.

The exec* family of functions also cause resource limits to be saved.

Returns:

0    Success.
-1   An error occurred (errno is set).

Errors:

EFAULT   The rlp argument points to an illegal address.
EINVAL   An invalid resource was specified, the new rlim_cur exceeds the new rlim_max, or the limit specified can’t be
setrlimit(), setrlimit64()

lowered because current usage is already higher than the limit.

**EPERM** The limit specified to `setrlimit()` would have raised the maximum limit value, and the effective user of the calling process isn’t the superuser.

**Classification:**

`setrlimit()` is POSIX 1003.1 XSI; `setrlimit64()` is Large-file support

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`brk()`, `execl()`, `execle()`, `execlelp()`, `execvp()`, `execve()`, `execvpe()`, `fork()`, `getdtablesize()`, `getrlimit()`, `getrlimit64()`, `malloc()`, `open()`, `signal()`, `sysconf()``ulimit` builtin command (see the entry for `ksh` in the *Utilities Reference*)
Synopsis:

```c
#include <netdb.h>

void setservent( int stayopen );
```

Arguments:

- `stayopen`  Nonzero if you don’t want the database to be closed after each call to `getservbyname()` or `getservbyport()`.

Library:

- `libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `setservent()` function opens and rewinds the network services database file. If the `stayopen` flag is nonzero, the network services database won’t be closed after each call to `getservbyname()` or `getservbyport()`.

Files:

- `/etc/services`
  
  Network services database file.

Classification:

- POSIX 1003.1

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No

*continued...*
setservent()

Safety

| Thread | No |

Caveats:

This function uses static data; if you need the data for future use, you should copy it before any subsequent calls overwrite it.

See also:

endservent(), getservbyname(), getservbyport(), getservent(),
servent

/etc/services in the Utilities Reference.
Synopsis:

```
#include <unistd.h>

pid_t setsid( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setsid()` function creates a new session with the calling process becoming the process group leader with no controlling terminal. The process group ID of the calling process is set to the process ID of the calling process. The calling process is the only process in the new process group, and is also the only process in the new session.

If the calling process is already a process group leader, a new session isn’t created and an error is returned.

Returns:

The new process group ID for the calling process, or -1 if an error occurred (`errno` is set).

Errors:

```
EPERM  The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.
```

Examples:

```
/*
 * You can only become a session leader if you are not
 * a process group leader that is the default for a
 * command run from the shell.
 */
```
setsid()

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    if( fork() )
    {
        if( setsid() == -1 )
            perror( "parent: setsid" );
        else
            printf( "parent: I am a session leader\n" );
    }
    else
    {
        if( setsid() == -1 )
            perror( "child: setsid" );
        else
            printf( "child: I am a session leader\n" );
    
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, getsid(), setpgid()
setsockopt()

Set options associated with a socket

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt( int s,
                int level,
                int optname,
                const void *optval,
                socklen_t optlen );
```

Arguments:

- `s`: The file descriptor of the socket that the option is to be applied on, as returned by `socket()`.

- `level`: The protocol layer that the option is to be applied to. In most cases, it's a socket-level option and is indicated by `SOL_SOCKET`.

- `optname`: The option for the socket file descriptor.

- `optval`: A pointer to the value of the option (in most cases, whether the option is to be turned on or off). If no option value is to be returned, `optval` may be `NULL`. Most socket-level options use an `int` parameter for `optval`. Others, such as the `SO_LINGER`, `SO_SNDTIMEO`, and `SO_RCVTIMEO` options, use structures that also let you get data associated with the option.

- `optlen`: A pointer to the length of the value of the option. This argument is a value-result parameter; you should initialize it to indicate the size of the buffer pointed to by `optval`. 
Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `setsockopt()` function sets the options associated with a socket.

*See* `getsockopt()` for more detailed information.

Returns:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success.</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (<code>errno</code> is set).</td>
</tr>
</tbody>
</table>

Errors:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid file descriptor <code>s</code>.</td>
</tr>
<tr>
<td>EDOM</td>
<td>Value was set out of range.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The address pointed to by <code>optval</code> isn’t in a valid part of the process address space.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>No <code>optval</code> value was specified.</td>
</tr>
<tr>
<td>ENOPROTOOPT</td>
<td>The option is unknown at the level indicated.</td>
</tr>
</tbody>
</table>

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

*continued...*
See also:

ICMP, IP, TCP, and UDP protocols

getsockopt(), socket()
Rewind the shadow password database file

Synopsis:

```c
#include <sys/types.h>
#include <shadow.h>

void setspent( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setspent()` function rewinds to the start of the shadow password database file. It’s provided for programs that make multiple lookups in the database (using the `getspnam()` call) to avoid opening and closing the shadow password database for each access.

Classification:

```
Unix
```

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

```
fgetspent() endspent(), getspnam(), getspent() putspent()
```
**setstate()**

*Reset the state of a pseudo-random number generator*

**Synopsis:**

```c
#include <stdlib.h>

char *setstate( const char *state );
```

**Arguments:**

- `state` A pointer to the state array that you want to use.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

---

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

Once the state of the pseudo-random number generator has been initialized, `setstate()` allows switching between state arrays. The array defined by the `state` argument is used for further random-number generation until `initstate()` is called or `setstate()` is called again. The `setstate()` function returns a pointer to the previous state array.

This function is used in conjunction with the following:

- `initstate()` Initialize the state of the pseudo-random number generator.
- `random()` Generate a pseudo-random number using a default state.
- `srandom()` Set the seed used by the pseudo-random number generator.

After initialization, you can restart a state array at a different point in one of two ways:
• Call `initstate()` with the desired seed, state array, and size of the array.
• Call `setstate()` with the desired state, then call `srandom()` with the desired seed. The advantage of using both of these functions is that the size of the state array doesn’t have to be saved once it’s initialized.

Returns:

A pointer to the previous state array, or NULL if an error occurred.

Examples:

See `initstate()`.

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`drand48()`, `initstate()`, `rand()`, `random()`, `srand()`, `srandom()`
Synopsis:

```c
#include <sys/time.h>

int settimeofday( const struct timeval *when, 
    void *not_used );
```

Arguments:

- `when` A pointer to a `timeval` structure that specifies the time that you want to set. The `struct timeval` contains the following members:
  - `long tv_sec` — the number of seconds since the start of the Unix Epoch.
  - `long tv_usec` — the number of microseconds.

- `not_used` This pointer must be NULL or the behavior of `settimeofday()` is unspecified. This argument is provided only for backwards compatibility.

Library:

```c
libc
```

Use the `–l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This function sets the time and date to the values stored in the structure pointed to by `when`.

Returns:

- 0, or -1 if an error occurred (`errno` is set).
Errors:

EFAULT An error occurred while accessing the *when* buffer.
EPERM The calling process doesn’t have superuser permissions.

Classification:

Legacy Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

The `settimeofday()` function is provided for compatibility with existing Unix code. You shouldn’t use it in new code — use `clock_settime()` instead.

See also:

`asctime()`, `asctime_r()`, `clock_gettime()`, `clock_settime()`, `ctime()`,
`ctime_r()`, `difftime()`, `gettimeofday()`, `gmtime()`, `gmtime_r()`,
`localtime()`, `localtime_r()`, `time()`
setuid()

Set the real, effective and saved user IDs

Synopsis:

```c
#include <unistd.h>

int setuid( uid_t uid );
```

Arguments:

- `uid` The user ID that you want to use for the process.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `setuid()` function lets the calling process set the real, effective and saved user IDs based on the following:

- If the process is the superuser, `setuid()` sets the real user ID, effective user ID and saved user ID to `uid`.
- If the process isn’t the superuser, but `uid` is equal to the real user ID or saved set-user ID, `setuid()` sets the effective user ID to `uid`; the real and saved user IDs aren’t changed.

A set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-UID.

If you wish to change only the effective user ID, and even if you are the superuser, you should consider using the `seteuid()` function.

The “superuser” is defined as any process with an effective user ID of `0`, or an effective user ID of `root`.
setuid()

Returns:

0 for success, or -1 if an error occurs (errno is set).

Errors:

EINVAL  The value of uid is out of range.

EPERM  The process isn’t the superuser, and uid doesn’t match the real user ID or saved set-user ID.

Examples:

/*
 * This process sets its userid to 0 (root)
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    uid_t ouid;
    ouid = getuid();
    if( setuid( 0 ) == -1 ) {
        perror( "setuid" );
        return EXIT_FAILURE;
    }
    printf( "userid %d switched to 0\n", ouid );
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
setuid()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, getuid(), setgid(), seteuid(), setgid()
setutent()

Synopsis:

```c
#include <utmp.h>

void setutent( void );
```

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `setutent()` function resets the input stream to the beginning of the file specified in `_PATH_UTMP`. Do this before each search for a new entry if you want the entire file to be examined.

Files:

`_PATH_UTMP`

Specifies the user information file.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

endutent(), getutent(), getutid(), getutline(), pututline(), utmp, utmpname()

login in the Utilities Reference


**setvbuf()**

Associate a buffer with a stream

**Synopsis:**

```c
#include <stdio.h>

int setvbuf( FILE *fp,
             char *buf,
             int mode,
             size_t size );
```

**Arguments:**

- **fp** The stream that you want to associate with a buffer.
- **buffer** NULL, or a pointer to the buffer; see below.
- **mode** How you want the stream to be buffered:
  - `_IOFBF` — input and output are fully buffered.
  - `_IOLBF` — output is line buffered (i.e. the buffer is flushed when a newline character is written, when the buffer is full, or when input is requested).
  - `_IONBF` — input and output are completely unbuffered.
- **size** The size of the buffer.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `setvbuf()` function associates a buffer with the stream designated by `fp`. If you want to call `setvbuf()`, you must call it after opening the stream, but before doing any reading, writing, or seeking.

If `buf` isn’t NULL, the buffer it points to is used instead of an automatically allocated buffer.
setvbuf()

Returns:

0  Success.
EINV   The mode argument isn’t valid.
ENOMEM   The buf argument is NULL, size isn’t 0, and there isn’t enough memory available to allocate a buffer.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  char *buf;
  FILE *fp;
  fp = fopen( "file", "r" );
  buf = malloc( 1024 );
  setvbuf( fp, buf, _IOFBF, 1024 );
  /* work with fp */
  ...
  fclose( fp );
  /* This is OUR buffer, so we have
   * to free it. Do that AFTER
   * you’ve closed the file.
   */
  free( buf );
  return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

Cancellation point  No

continued…
setvbuf() © 2007, QNX Software Systems GmbH & Co. KG.

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fopen(), setbuf()
Deallocate a block of memory

Synopsis:

```c
#include <malloc.h>

void _sfree( void *ptr,
            size_t size );
```

Arguments:

- **ptr**: NULL, or a pointer to the block of memory that you want to free.
- **size**: The number of bytes to deallocate.

Library:

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

When the value of the argument `ptr` is NULL, the `_sfree()` function does nothing; otherwise, the `_sfree()` function deallocates `size` bytes of memory located by the argument `ptr`, which was previously returned by a call to the appropriate version of `_salloc()` or `_smalloc()`. After the call, the freed block is available for allocation.

Classification:

QNX Neutrino
## _sfree()

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Caveats:

Calling _sfree() on a pointer already deallocated by a call to _sfree() could corrupt the memory allocator’s data structures.

The *size must* match the size of the allocated block.

### See also:

`calloc(), free(), realloc(), _scalloc(), _smalloc(), _srealloc()`
**shm_ctl()**

Give special attributes to a shared memory object

**Synopsis:**

```c
#include <sys/mman.h>

int shm ctl( int fd,
    int flags,
    _uint64 paddr,
    _uint64 size );
```

**Arguments:**

- **fd** The file descriptor that’s associated with the shared memory object, as returned by `shm_open()`.
- **flags** One or more of the following bits, defined in `<sys/mman.h>`:
  - SHMCTL_ANON — allocate anonymous memory.
  - SHMCTL_PHYS — use physical address, or allocate physically contiguous memory if used with SHMCTL_ANON.
  - SHMCTL_GLOBAL — a hint that any mapping to the object could be global across all processes.
  - SHMCTL_PRIV — a hint that a mapping of this object may require privileged access.
  - SHMCTL_LOWERPROT — a hint that the system may map this object in such a way that it trades lower memory protection for better performance.
  - SHMCTL.LAZYWRITE — a hint that a mapping of this object could use lazy-writing mechanisms.

Some of the bits have specific meanings for an ARM processor. For more information, see “Flags for ARM processors,” below.

- **paddr** A physical address to assign to the object, if you set SHMCTL_PHYS in flags.
- **size** The new size of the object, in bytes, regardless of ANON/PHYS flag.
Library:

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

Description:

The \texttt{shm\_ctl()} function modifies the attributes of the shared memory object identified by the handle, \texttt{fd}. This handle is the value returned by \texttt{shm\_open()}.

- The combination \texttt{SHMCTL\_ANON | SHMCTL\_PHYS} has the same behavior as for \texttt{mmap():} it indicates that you want physically contiguous RAM to be allocated for the object.

- On ARM targets, once you’ve called \texttt{shm\_ctl()} for a shared memory object, you can’t resize it. You must unmap and unlink the shared object, and then recreate it.

The \texttt{shm\_ctl\_special()} function is similar to \texttt{shm\_ctl()}, but has an additional, processor-specific argument.

Flags for ARM processors

For an ARM processor, the behavior for different combinations of \texttt{flags} arguments is as follows:

\texttt{SHMCTL\_ANON}

Create a shared memory object using anonymous memory (i.e. zero-filled, dynamically allocated RAM). This memory isn’t guaranteed to be physically contiguous.

When you \texttt{mmap()}, it’s mapped into the normal process address space, so the mapping is subject to the 32M address space limit on ARM.
SHMCTL_ANON | SHMCTL_PHYS

Same as SHMCTL_ANON, except that the memory allocated is physically contiguous.

When you mmap(), it’s mapped into the normal process address space, so the mapping is subject to the 32M address space limit on ARM.

SHMCTL_ANON | SHMCTL_GLOBAL

Allocate memory as for SHMCTL_ANON.

When you mmap(), it’s mapped at a globally visible address outside the regular process address space. This allows the object to be up to 1GB in size.

If more than one process maps the object, it appears at the same virtual address in all processes that map it. This virtual address is assigned at the time the first mmap() of the object is performed.

The mmap() call must map the whole object (i.e. offset must be 0, and len must be the size of the object set by shm_ctl()).

The mappings are protected such that only those processes that have mapped the object can access it. Any other process that attempts to access the (globally visible) virtual address fails. This impacts context switches to and from the mapping process because MMU page tables must be modified to grant and remove the access permissions on each context switch.

The additional cost of this manipulation during context switches includes modifying one Level 1 table entry for each megabyte of the mapping, followed by a TLB flush.

SHMCTL_ANON | SHMCTL_GLOBAL | SHMCTL_PRIV

Same as SHMCTL_ANON | SHMCTL_GLOBAL, except that no per-process protection is performed. Instead, the access permissions for the (global) mapping are set, so that only processes that have I/O privileges (i.e. have called ThreadCtl() with a command of NTO_TCTL_IO) can access the mappings.

This avoids the performance impact during context switches, but allows potential access by any process that has I/O.
privileges. I/O privileges are normally required only for driver or other system processes, so this combination provides some protection from potential access by normal user processes.

**SHMCTL_ANON | SHMCTL_GLOBAL | SHMCTL_LOWERPROT**

Same as SHMCTL_ANON | SHMCTL_GLOBAL, except that no per-process protection is performed; any process can access the mappings.

This avoids the performance impact during context switches and also avoids requiring I/O privileges to access the mappings. However, there is no protection from potential access by other processes.

**SHMCTL_PHYS**

Create an object to map the physical address specified by the offset parameter to the `shm_ctl()` call.

When you `mmap()`, it’s mapped at a globally visible address outside the normal process address space. This allows the object to be up to 1GB in size.

All mappings are forced to PROT_NOCACHE to avoid cache aliases (due to the ARM virtually indexed cache)

Each process that maps, the object gets a new mapping.

The mappings are protected such that only those processes that have mapped the object can access it. Any other process that attempts to access the (globally visible) virtual address fail.

This impacts context switches to and from the mapping process because MMU page tables must be modified to grant and remove the access permissions on each context switch.

The additional cost of this manipulation during context switches includes modifying one Level 1 table entry for each megabyte of the mapping, followed by a TLB flush.

**SHMCTL_PHYS | SHMCTL_PRIV**

Same as SHMCTL_PHYS, except that mappings aren’t protected. Instead, access is allowed only by processes with I/O privileges.
This avoids the performance impact during context switches.

**SHMCTL_PHYS | SHMCTL_LOWERPROT**

Same as SHMCTL_PHYS, except that mappings aren’t protected. Access is allowed for any process.

**SHMCTL_PHYS | SHMCTL_GLOBAL**

Same as SHMCTL_PHYS, except that all processes that map the object use the same virtual address. This virtual address is assigned by the first `mmap()` call to map the object.

Because all processes use the same (globally visible) virtual address, these mappings aren’t forced to PROT_NOCACHE. If you need uncached behavior, specify PROT_NOCACHE when you call `mmap()`.

The mappings are protected such that only those processes that have mapped the object can access it. Any other process that attempts to access the (globally visible) virtual address will fault. This impacts context switches to and from the mapping process because MMU page tables must be modified to grant and remove the access permissions on each context switch.

The additional cost of this manipulation during context switches includes modifying one Level 1 table entry for each megabyte of the mapping, followed by a TLB flush.

**SHMCTL_PHYS | SHMCTL_GLOBAL | SHMCTL_PRIV**

Same as SHMCTL_PHYS | SHMCTL_GLOBAL, except that mappings aren’t protected. Instead, access is allowed only by processes with I/O privileges.

This avoids the performance impact during context switches.

**SHMCTL_PHYS | SHMCTL_GLOBAL | SHMCTL_LOWERPROT**

Same as SHMCTL_PHYS | SHMCTL_GLOBAL, except that mappings aren’t protected. Instead, access is allowed by any process.

This avoids the performance impact during context switches.
shm_ctl()

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EINVAL An invalid combination of flags was specified, or the shared memory object is already “special.”

Examples:

The following examples go together. Run sharephyscreator, followed by sharephysuser.

The sharephyscreator process maps in an area of physical memory and then overlays it with a shared memory object. The sharephysuser process opens that shared memory object in order to access the physical memory.

```c
/*
 * sharephyscreator.c
 *
 * This maps in an area of physical memory and then overlays it with a shared memory object. This way, another process can open that shared memory object in order to access the physical memory. The other process in this case is sharephysuser.
 * Note that the size and address that you pass to shm_ctl() must be even multiples of the page size (sysconf(_SC_PAGE_SIZE)).
 * For VGA color text mode video memory:
 * sharephyscreator /wally b8000
 * Note that for VGA color text mode video memory, each character is followed by an attribute byte. Here we just use a space.
 */

#include <errno.h>
#include <fcntl.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/stat.h>

char *progname = "sharephyscreator";
```
main( int argc, char *argv[] )
{
    char *text = "Hello world!";
    int fd, memsize;
    char *ptr, *name;
    uint64_t physaddr;

    if ( argc != 3 ) {
        printf( "Use: sharephyscreator shared_memory_object_name \n physical_address_in_hex\n" );
        printf( "Example: sharephyscreator wally b8000\n" );
        exit( EXIT_FAILURE );
    }
    name = argv[1];
    physaddr = atoh(argv[2]);
    memsize = sysconf( _SC_PAGE_SIZE ); /* this should be enough
          for our string */

    /* map in the physical memory */
    ptr = mmap_device_memory( 0, memsize, PROT_READ|PROT_WRITE, 0, physaddr );
    if ( ptr == MAP_FAILED ) {
        printf( "%s: mmap_device_memory for physical address %llx failed: %s
", 
            progname, physaddr, strerror(errno) );
        exit( EXIT_FAILURE );
    }

    /* open the shared memory object, create it if it doesn't exist */
    fd = shm_open( name, O_RDWR | O_CREAT, 0 );
    if ( fd == -1 ) {
        printf( "%s: error creating the shared memory object '%s': %s\n", 
            progname, name, strerror(errno) );
        exit( EXIT_FAILURE );
    }

    /* overlay the shared memory object onto the physical memory */
    if ( shm_ctl(fd, SHMCTL_PHYS, physaddr, memsize) == -1 ) {
        printf( "%s: shm_ctl failed: %s\n", 
                progname, strerror(errno) );
        close(fd);
        munmap(ptr, memsize);
        shm_unlink(name);
        exit(EXIT_FAILURE);
    }

    strcpy(ptr, text); /* write to the shared memory */

    printf( "%s: Physical memory mapped in, shared memory overlayed onto it.\n" 
            "%s: Wrote '%s' to physical memory.\n" 
            "%s: Sleeping for 20 seconds. While this program is sleeping\n" 
            "%s: run 'sharephysuser %s %d'.\n", 
            progname, progname, ptr, progname, name, 
            strlen(text)+1 );
    sleep(20);

    printf( "%s: Woke up. Cleaning up and exiting ...\n", progname );
    close(fd);
    munmap(ptr, memsize);
    shm_unlink(name);
}
The following is meant to be run with `sharephyscreator`.

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/stat.h>

char *progname = "sharephysuser";

main( int argc, char *argv[] )
{
    int fd, len, i;
    char *ptr, *name;
    if ( argc != 3 ) {
        fprintf( stderr, "Use: sharephysuser shared_memory_object_name length
Example: sharephysuser wally 49
" );
        exit( EXIT_FAILURE );
    }
    name = argv[1];
    len = atoi( argv[2] );
    /* open the shared memory object */
    fd = shm_open( name, O_RDWR, 0 );
    if ( fd == -1 ) {
        fprintf( stderr, "%s: error opening the shared memory object '%s': %s
", progname, name, strerror(errno) );
        exit( EXIT_FAILURE );
    }
    /* get a pointer to a piece of the shared memory, note that we only map in the amount we need to */
    ptr = mmap( 0, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0 );
    if ( ptr == MAP_FAILED ) {
        fprintf( stderr, "%s: mmap failed: %s
", progname, strerror(errno) );
        exit( EXIT_FAILURE );
    }
    printf( "%s: reading the text: ", progname );
```
for ( i = 0; i < len; i++ )
    printf( "\x", ptr[i] );
    printf( "\n" );
    close( fd );
    munmap( ptr, len );
}

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mmap(), munmap(), mprotect(), shm_ctl_special(), shm_open(), shm_unlink(), ThreadCtl()
**shm_ctl_special()**

Give special attributes to a shared memory object

**Synopsis:**

```c
#include <sys/mman.h>

int shm_ctl_special( int fd,
    int flags,
    _uint64 paddr,
    _uint64 size,
    unsigned special );
```

**Arguments:**

- **fd**
  The file descriptor that’s associated with the shared memory object, as returned by `shm_open()`.

- **flags**
  One or more of the following bits, defined in `<sys/mman.h>`:
  - SHMCTL_ANON — allocate anonymous memory.
  - SHMCTL_PHYS — use physical address, or allocate physically contiguous memory if used with SHMCTL_ANON.
  - SHMCTL_GLOBAL — a hint that any mapping to the object could be global across all processes.
  - SHMCTL_PRIV — a hint that a mapping of this object may require privileged access.
  - SHMCTL_LOWERPROT — a hint that the system may map this object in such a way that it trades lower memory protection for better performance.
  - SHMCTL_LAZYWRITE — a hint that a mapping of this object could use lazy-writing mechanisms.

Some of the bits have specific meanings for different processors. For more information, see the documentation for `shm_ctl()`.

- **paddr**
  A physical address to assign to the object, if you set SHMCTL_PHYS in flags.
The new size of the object, in bytes, regardless of ANON/PHYS flag.

Process-specific flags.
This argument is currently used only on SH4 platforms.
On SH4 7760, it controls the space attribute bits of the UTLB (see section 6.3.1 of 7760 hardware manual).

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The *shm_ctl_special()* function modifies the attributes of the shared memory object identified by the handle, *fd*. This handle is the value returned by *shm_open()*.

The *shm_ctl_special()* function is similar to *shm_ctl()* but has an additional processor-specific argument, *special*. Calling *shm_ctl_special()* with a value of 0 for *special* is equivalent to calling *shm_ctl()*.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Errors:

EINVAL An invalid combination of flags was specified, or the shared memory object is already “special.”
Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`mmap()`, `munmap()`, `mprotect()`, `shm_ctl()`, `shm_open()`, `shm_unlink()`, `ThreadCtl()`
shmem_open()
Open a shared memory object

Synopsis:
#include <fcntl.h>
#include <sys/mman.h>

int shmem_open( const char * name,
                int iflag,
                mode_t mode );

Arguments:
name The name of the shared memory object that you want to open; see below.

iflag A combination of the following bits (defined in <fcntl.h>):
- O_RDONLY — open for read access only.
- O_RDWR — open for read and write access.
- O_CREAT — if the shared memory object exists, this flag has no effect, except as noted under O_EXCL below. Otherwise, the shared memory object is created, and its permissions are set in accordance with the value of mode and the file mode creation mask of the process.
- O_EXCL — if O_EXCL and O_CREAT are set, then shmem_open() fails if the shared memory segment exists. The check for the existence of the shared memory object, and the creation of the object if it doesn’t exist, are atomic with respect to other processes executing shmem_open(), naming the same shared memory object with O_EXCL and O_CREAT set.
- O_TRUNC — if the shared memory object exists, and it’s successfully opened O_RDWR, the object is truncated to zero length and the mode and owner are unchanged.

mode The permission bits for the memory object are set to the value of mode, except those bits set in the process’s file creation mask. For more information, see umask(), and “Access permissions” in the documentation for stat().
**shm_open()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Library:**

libraries:

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `shm_open()` function returns a file descriptor that’s associated with the shared “memory object” specified by `name`. This file descriptor is used by other functions to refer to the shared memory object (for example, `mmap()`, `mprotect()`). The FD_CLOEXEC flag in `fcntl()` is set for this file descriptor.

The `name` argument is interpreted as follows:

```
<table>
<thead>
<tr>
<th>name</th>
<th>Pathname space entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry</td>
<td>CWD/entry</td>
</tr>
<tr>
<td>/entry</td>
<td>/dev/shmem/entry</td>
</tr>
<tr>
<td>entry/newentry</td>
<td>CWD/entry/newentry</td>
</tr>
<tr>
<td>/entry/newentry</td>
<td>/entry/newentry</td>
</tr>
</tbody>
</table>
```

where `CWD` is the current working directory for the program at the point that it calls `mq_open()`.

If you want to open a shared memory object on another node, you have to specify the name as `/net/node/shmem_location`.

The state of the shared memory object, including all data associated with it, persists until the shared memory object is unlinked and all other references are gone.

**Returns:**

A nonnegative integer, which is the lowest numbered unused file descriptor, or `-1` if an error occurred (`errno` is set).
Errors:

EACCES  Permission to create the shared memory object is denied.
The shared memory object exists and the permissions specified by \texttt{oflag} are denied, or \texttt{O\_TRUNC} is specified and write permission is denied.

EEXIST  \texttt{O\_CREATE} and \texttt{O\_EXCL} are set, and the named shared memory object already exists.

EINTR  The \texttt{shm\_open()} call was interrupted by a signal.

ELOOP  Too many levels of symbolic links or prefixes.

EMFILE  Too many file descriptors are currently in use by this process.

ENAMETOOLONG

The length of the \texttt{name} argument exceeds \texttt{NAME\_MAX}.

ENOFILE  Too many shared memory objects are currently open in the system.

ENOENT  \texttt{O\_CREATE} isn’t set, and the named shared memory object doesn’t exist, or \texttt{O\_CREATE} is set and either the name prefix doesn’t exist or the \texttt{name} argument points to an empty string.

ENOSPC  There isn’t enough space to create the new shared memory object.

ENOSYS  The \texttt{shm\_open()} function isn’t supported by this implementation.

Examples:

This example sets up a shared memory object, but doesn’t really do anything with it:
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <sys/mman.h>

int main( int argc, char** argv )
{
    int fd;
    unsigned* addr;

    /*
    * In case the unlink code isn’t executed at the end
    */
    if( argc != 1 ) {
        shm_unlink( "/bolts" );
        return EXIT_SUCCESS;
    }

    /* Create a new memory object */
    fd = shm_open( "/bolts", O_RDWR | O_CREAT, 0777 );
    if( fd == -1 ) {
        fprintf( stderr, "Open failed:%s\n",
            strerror( errno ) );
        return EXIT_FAILURE;
    }

    /* Set the memory object’s size */
    if( ftruncate( fd, sizeof( *addr ) ) == -1 ) {
        fprintf( stderr, "ftruncate: %s\n",
            strerror( errno ) );
        return EXIT_FAILURE;
    }

    /* Map the memory object */
    addr = mmap( 0, sizeof( *addr ),
        PROT_READ | PROT_WRITE,
        MAP_SHARED, fd, 0 );
    if( addr == MAP_FAILED ) {
        fprintf( stderr, "mmap failed: %s\n",
            strerror( errno ) );
        return EXIT_FAILURE;
    }

    printf( "Map addr is 0x%08x\n", addr );

    /* Write to shared memory */
shm_open()

*addr = 1;

/*
 * The memory object remains in
 * the system after the close
 */
close(fd);

/*
 * To remove a memory object
 * you must unlink it like a file.
 * This may be done by another process.
 */
shm_unlink("/bolts");

return EXIT_SUCCESS;
}

This example uses a shared memory object to share data with a forked process:

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

main(int argc, char * argv[])
{
  int fd;
  unsigned *addr;

  /* In case the unlink code isn’t executed at the end */
  if (argc != 1) {
    shm_unlink("/bolts");
    return EXIT_SUCCESS;
  }

  /* Create a new memory object */
  fd = shm_open("/bolts", O_RDWR | O_CREAT, 0777);
  if (fd == -1) {
    fprintf(stderr, "Open failed : %s\n",
            strerror(errno));
    return EXIT_FAILURE;
  }

  *addr = 1;

  /* The memory object remains in
   * the system after the close
   */
  close(fd);

  /* To remove a memory object
   * you must unlink it like a file.
   * This may be done by another process.
   */
  shm_unlink("/bolts");

  return EXIT_SUCCESS;
}
/* Set the memory object’s size */
if (ftruncate(fd, sizeof(*addr)) == -1) {
    fprintf(stderr, "ftruncate : %s\n", strerror(errno));
    return EXIT_FAILURE;
}

/* Map the memory object */
addr = mmap(0, sizeof(*addr), PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0);
if (addr == MAP_FAILED) {
    fprintf(stderr, "mmap failed:%s\n", strerror(errno));
    return EXIT_FAILURE;
}

printf("Map addr is %6.6X\n", addr);
printf("Press break to stop.\n");
sleep(3); /* So you can read above message */

/*
 * We unlink so object goes away on last close.
 */
shm_unlink("/bolts");

*addr = ‘0’;
if (fork())
    for (;;)
        if (*addr == ‘0’)
           putc(*addr = ‘1’, stderr);
        else
            sched_yield();
else
    for (;;)
        if (*addr == ‘1’)
           putc(*addr = ‘0’, stderr);
        else
            sched_yield();
return EXIT_SUCCESS;
}
shm_open()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fcntl(), ftruncate(), mmap(), munmap(), mprotect(), open(), shm_ctl(),
shm_ctl_special(), shm_unlink(), sysconf()
**shm_unlink()**

Remove a shared memory object

**Synopsis:**

```c
#include <sys/mman.h>

int shm_unlink( const char * name );
```

**Arguments:**

- `name` The name of the shared memory object that you want to remove.

**Library:**

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `shm_unlink()` function removes the name of the shared memory object specified by `name`. After removing the name, you can’t use `shm_open()` to access the object.

This function doesn’t affect any references to the shared memory object (i.e. file descriptors or memory mappings). If more than one reference to the shared memory object exists, then the link count is decremented, but the shared memory segment isn’t actually removed until you remove all open and map references to it.

**Returns:**

- 0 Success.
- -1 An error occurred (errno is set).

**Errors:**

- `EACCES` Permission to unlink the shared memory object is denied.
- `ELOOP` Too many levels of symbolic links or prefixes.
shm.unlink()

ENAMETOOLONG
The length of the name argument exceeds NAME_MAX.
ENOENT The named shared memory object doesn’t exist, or the name argument points to an empty string.
ENOSYS The shm.unlink() function isn’t supported by this implementation.

Examples:
See shm.open().

Classification:
POSIX 1003.1 SHM

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
mmap(), munmap(), mprotect(), shm_ctl(), shm_ctl_special(), shm.open()
**shutdown()**

Shut down part of a full-duplex connection

**Synopsis:**

```c
#include <sys/socket.h>

int shutdown( int s,
              int how );
```

**Arguments:**

- `s` A descriptor for the socket, as returned by `socket()`.
- `how` How you want to shut down the connection:
  
  **If how is:** The TCP/IP manager won’t allow:
  
<table>
<thead>
<tr>
<th>How</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHUT_RD</td>
<td>Further receives</td>
</tr>
<tr>
<td>SHUT_WR</td>
<td>Further sends</td>
</tr>
<tr>
<td>SHUT_RDWR</td>
<td>Further sends and receives</td>
</tr>
</tbody>
</table>

**Library:**

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

**Description:**

The `shutdown()` call shuts down all or part of a full-duplex connection on the socket associated with `s`.

**Returns:**

- `0` Success.
- `-1` An error occurred (`errno` is set).
Errors:

EBADF Invalid descriptor s.

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

close(), connect(), socket()
**sigaction()**

Examine or specify the action associated with a signal

**Synopsis:**

```c
#include <signal.h>

int sigaction( int sig,
               const struct sigaction * act,
               struct sigaction * oact );
```

**Arguments:**

- **sig**
  The signal number (defined in `<signal.h>`). For more information, see “POSIX signals” in the documentation for `SignalAction()`.

- **act**
  NULL, or a pointer to a `sigaction` structure that specifies how you want to modify the action for the given signal. For more information about this structure, see below.

- **oact**
  NULL, or a pointer to a `sigaction` structure that the function fills with information about the current action for the signal.

**Library:**

- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

You can use `sigaction()` to examine or specify (or both) the action that’s associated with a specific signal:

- If `act` isn’t NULL, the specified signal is modified.
- If `oact` isn’t NULL, the previous action is stored in the structure it points to.

The structure `sigaction` contains the following members:
void (*sa_handler)();

Address of a signal handler or action for nonqueued signals.

void (*sa_sigaction)(int signo, siginfo_t* info, void* other);

Address of a signal handler or action for queued signals.

sigset_t sa_mask

An additional set of signals to be masked (blocked) during execution of the signal-catching function.

int sa_flags

Special flags to affect behavior of the signal:

- SA_NOCLDSTOP is only used when the signal is SIGCHLD. It tells the system not to generate a SIGCHLD on the parent for children who stop via SIGSTOP.
- SA_SIGINFO tells the OS to queue this signal. The default is not to queue a signal delivered to a process. If a signal isn’t queued, setting the same signal multiple times on a process or thread before it runs results in only the last signal’s being delivered. If you set SA_SIGINFO, the signals are queued and they’re all delivered.

The sa_handler and sa_sigaction members of act are implemented as a union and share common storage. They differ only in their prototypes, with sa_handler being used for POSIX 1003.1a signals and sa_sigaction being used for POSIX 1003.1b queued realtime signals. The values stored using either name can be one of:

function

The address of a signal catching function. See below for details.

SIG_DFL

This sets the signal to the default action:
SIGCHLD, SIGIO, SIGURG and SIGWINCH — ignore the signal (SIG_IGN).

SIGSTOP — stop the process.

SIGCONT — continue the program.

All other signals — kill the process.

SIG_IGN

This ignores the signal. Setting SIG_IGN for a signal that’s pending discards all pending signals, whether it’s blocked or not. New signals are discarded. If you ignore SIGCHLD, your process’s children don’t enter the zombie state and you’re unable to wait on their death using wait() or waitpid().

The function member of sa_handler or sa__sigaction is always invoked with the following arguments:

\[
\text{void \ handler(int \ signo, siginfo_t *info, void *other)}
\]

If you have an old-style signal handler of the form:

\[
\text{void \ handler(int \ signo)}
\]

the extra arguments are still placed by the kernel, but the function simply ignores them.

While in the handler, signo is masked, preventing nested signals of the same type. In addition, any signals set in the sa_mask member of act are also ORed into the mask. When the handler returns through a normal return, the previous mask is restored and any pending and now unmasked signals are acted on. You return to the point in the program where it was interrupted. If the thread was blocked in the kernel when the interruption occurred, the kernel call returns with an EINTR (see ChannelCreate() and SyncMutexLock() for exceptions to this).

The siginfo_t structure of the function in sa_handler or sa__sigaction contains at least the following members:

\[
\text{int \ si_signo \quad The \ signal \ number, \ which \ should \ match \ the \ signo \ argument \ to \ the \ handler.}
\]
int si_code  
A signal code, provided by the generator of the signal:

- SI_USER — the kill() function generated the signal.
- SI_QUEUE — the sigqueue() function generated the signal.
- SI_TIMER — a timer generated the signal.
- SI_ASYNCIO — asynchronous I/O generated the signal.
- SI_MESGQ — POSIX (not QNX) messages queues generated the signal.

union sigval si_value  
A value associated with the signal, provided by the generator of the signal.

You can’t ignore or catch SIGKILL or SIGSTOP.

Signal handlers and actions are defined for the process and affect all threads in the process. For example, if one thread ignores a signal, then all threads ignore the signal.

You can target a signal at a thread, process, or process group (see SignalKill()). When targeted at a process, at most one thread receives the signal. This thread must have the signal unblocked (see SignalProcmask()) to be a candidate for receiving it. All synchronously generated signals (e.g. SIGSEGV) are always delivered to the thread that caused them.

If you use longjmp() to return from a signal handler, the signal remains masked. You can use siglongjmp() to restore the mask to the state saved by a previous call to sigsetjmp().
sigaction()

© 2007, QNX Software Systems GmbH & Co. KG.

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EAGAIN Insufficient system resources are available to set up the signal’s action.

EFAULT A fault occurred trying to access the buffers provided.

EINVAL The signal signo isn’t valid.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main( void )
{
    extern void handler();
    struct sigaction act;
    sigset_t set;

    sigemptyset( &set );
    sigaddset( &set, SIGUSR1 );
    sigaddset( &set, SIGUSR2 );

    /*
     * Define a handler for SIGUSR1 such that when
     * entered both SIGUSR1 and SIGUSR2 are masked. *
     */
    act.sa_flags = 0;
    act.sa_mask = set;
    act.sa_handler = &handler;
    sigaction( SIGUSR1, &act, NULL );

    kill( getpid(), SIGUSR1 );

    /* Program will terminate with a SIGUSR2 */
    return EXIT_SUCCESS;
}

void handler( signo )
```

2958 C Library — S September 10, 2007
{  
    static int first = 1;  
    
    if( first ) {  
        first = 0;  
        kill( getpid(), SIGUSR1 ); /* Prove signal masked */  
        kill( getpid(), SIGUSR2 ); /* Prove signal masked */  
    }  
}  

/*  
* - SIGUSR1 is set from main(), handler() is called.  
* - SIGUSR1 and SIGUSR2 are set from handler().  
* - however, signals are masked until we return to main().  
* - returning to main() unmasks SIGUSR1 and SIGUSR2.  
* - pending SIGUSR1 now occurs, handler() is called.  
* - pending SIGUSR2 now occurs. Since we don’t have  
*   a handler for SIGUSR2, we are killed.  
*/

Classification:  

POSIX 1003.1

Safety  

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:  

errno, kill(), raise(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), SignalAction(), SignalKill(), sigpending(), sigprocmask()
**sigaddset()**

Add a signal to a set

Synopsis:

```c
#include <signal.h>

int sigaddset( sigset_t *set,
               int signo );
```

Arguments:

- **set**: A pointer to the `sigset_t` object that you want to add the signal to.
- **signo**: The signal that you want to add. For more information, see “POSIX signals” in the documentation for `SignalAction()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigaddset()` function adds `signo` to the set pointed to by `set`.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).

Errors:

- `EINVAL` The signal `signo` isn’t valid.

Examples:

See `sigemptyset()`.
sigaddset()

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

kill(), raise(), sigaction(), sigdelset(), sigemptyset(), sigfillset(),

sigismember(), signal(), sigpending(), sigprocmask()
**sigblock()**

*Add to the mask of signals to block*

**Synopsis:**

```c
#include <unix.h>

int sigblock( int mask );
```

**Arguments:**

- `mask` - A bitmask of the signals that you want to block.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigblock()` function adds the signals specified in `mask` to the set of signals currently being blocked from delivery. Signals are blocked if the appropriate bit in `mask` is a 1; the macro `sigmask()` is provided to construct the mask for a given signum. The `sigblock()` returns the previous mask. You can restore the previous mask by calling `sigsetmask()`.

In normal usage, a signal is blocked using `sigblock()`. To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there’s no work to be done, and the process pauses awaiting work by using `sigpause()` with the mask returned by `sigblock()`.

It isn’t possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

**Returns:**

The previous set of masked signals.
sigblock()  

Classification:  
Unix  

Safety  
<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:  
Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

See also:  
kill(), sigaction(), sigmask(), signal(), sigpause(), sigprocmask(), sigsetmask(), sigunblock()
sigdelset() — Delete a signal from a set

Synopsis:

```c
#include <signal.h>

int sigdelset( sigset_t *set, int signo );
```

Arguments:

- `set` A pointer to the `sigset_t` object that you want to remove the signal from.
- `signo` The signal that you want to remove. For more information, see “POSIX signals” in the documentation for `SignalAction()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigdelset()` function deletes `signo` from the set pointed to by `set`.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).

Errors:

- `EINVAL` The signal `signo` isn’t valid.
Examples:
See *sigemptyset()*.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*kill()*,*raise()*,*sigaction()*,*sigaddset()*,*sigemptyset()*,*sigfillset()*,*sigismember()*,*signal()*,*sigpending()*,*sigprocmask()*
Initialize a set to contain no signals

Synopsis:

```c
#include <signal.h>

int sigemptyset( sigset_t *set );
```

Arguments:

- `set` A pointer to the `sigset_t` object that you want to initialize.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigemptyset()` function initializes `set` to contain no signals.

Returns:

- 0 Success.
- -1 An error occurred (`errno` is set).

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void print( sigset_t set, int signo )
{
    printf( "Set %8.8lx. Signal %d is ", set, signo );
    if( sigismember( &set, signo ) )
        printf( "a member.\n" );
    else
        printf( "not a member.\n" );
}

int main( void )
{
```
sigemptyset()

```
sigset_t set;

sigemptyset(&set);
print(set, SIGINT);

sigfillset(&set);
print(set, SIGINT);

sigdelset(&set, SIGINT);
print(set, SIGINT);

sigaddset(&set, SIGINT);
print(set, SIGINT);

return EXIT_SUCCESS;
```
Structure that describes an event

Synopsis:
```
#include <sys/siginfo.h>

union sigval {
    int sival_int;
    void   * sival_ptr;
};
```

The `sigevent` structure is complicated; see below.

Description:

This structure describes an event. The `int sigev_notify` member indicates how the notification is to occur, as well as which of the other members are used:

<table>
<thead>
<tr>
<th><code>sigev_notify</code></th>
<th><code>sigev_signo</code></th>
<th><code>sigev_coid</code></th>
<th><code>sigev_priority</code></th>
<th><code>sigev_code</code></th>
<th><code>sigev_value</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGEV_INTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_NONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_PULSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_SIGNAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_SIGNAL_CODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_SIGNAL_THREAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_THREAD</td>
<td>(special — see below)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGEV_UNBLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `<sys/siginfo.h>` file also defines some macros to make initializing the `sigevent` structure easier. All the macros take a pointer to a `sigevent` structure as their first argument, `event`, and set the `sigev_notify` member to the appropriate value.
These macros are QNX Neutrino extensions.

The library might use hidden bits in the *sigev_notify* member of the *sigevent* structure. If you ever test the value of this member after creating a *sigevent*, don’t compare it directly to a value; use the *SIGEV_GET_TYPE()* macro. For example, instead of:

```c
if( my_event.sigev_notify == SIGEV_PULSE)
```

use:

```c
if( SIGEV_GET_TYPE(&my_event) == SIGEV_PULSE)
```

### SIGEV_INTR

Send an interrupt notification to a specific thread. No other fields in the structure are used.

The initialization macro is:

```c
SIGEV_INTR_INIT( &event )
```

### SIGEV_NONE

Don’t send any notification. No other fields in the structure are used.

The initialization macro is:

```c
SIGEV_NONE_INIT( &event )
```

### SIGEV_PULSE

Send a pulse. The following fields are used:

```c
int sigev_coid
```

The connection ID. This should be attached to the channel with which the pulse will be received.

```c
short sigev_priority
```

The priority of the pulse.
If you want the thread that receives the pulse to run at the initial priority of the process, set `sigev_priority` to `SIGEV_PULSE_PRIO_INHERIT`.

**short sigev_code**

A code to be interpreted by the pulse handler. Although `sigev_code` can be any 8-bit signed value, you should avoid `sigev_code` values less than zero in order to avoid conflict with kernel or pulse codes generated by a QNX manager. These codes all start with `_PULSE_CODE_` and are defined in `<sys/neutrino.h>`; for more information, see the documentation for the `_pulse` structure. A safe range of pulse values is `_PULSE_CODE_MINAVAIL` to `_PULSE_CODE_MAXAVAIL`.

**void *sigev_value.sival_int**

A 32-bit value to be interpreted by the pulse handler.

The initialization macro is:

```c
SIGEV_PULSE_INIT(&event, coid, priority, code, value)
```

**SIGEV_SIGNAL**

Send a signal to a process. The following fields are used:

**int sigev_signo**

The signal to raise. This must be in the range from 1 through `NSIG − 1`.

The initialization macro is:

```c
SIGEV_SIGNAL_INIT(&event, signal)
```

If you need to set the `sigev_value` for a SIGEV_SIGNAL event (for example if `SA_SIGINFO` is set), you can use this macro:

```c
SIGEV_SIGNAL_VALUE_INIT(&event, signal, value)
```
SIGEV_SIGNAL_CODE

This is similar to SIGEV_SIGNAL, except that SIGEV_SIGNAL_CODE also includes a value and a code. The following fields are used:

```
int sigev_signo
   The signal to raise. This must be in the range from 1 through NSIG – 1.
short sigev_code
   A code to be interpreted by the signal handler. This must be in the range from SI_MINAVALID through SI_MAXAVALID.
void *sigev_value.sival_int
   A 32-bit value to be interpreted by the signal handler.
```

The initialization macro is:

```
SIGEV_SIGNAL_CODE_INIT(&event, signal, value, code)
```

SIGEV_SIGNAL_THREAD

Send a signal to a specific thread. The following fields are used:

```
int sigev_signo
   The signal to raise. This must be in the range from 1 through NSIG – 1.
short sigev_code
   A code to be interpreted by the signal handler. This must be in the range from SI_MINAVALID through SI_MAXAVALID.
void *sigev_value.sival_int
   A 32-bit value to be interpreted by the signal handler.
```

The initialization macro is:

```
SIGEV_SIGNAL_THREAD_INIT(&event, signal, value, code)
```
SIGEVTHREAD

Create a new thread.

We don’t recommend using this type of event. Pulses are more efficient.

The following fields are used:

\[
\text{void} (*\text{sigev}._\text{notify}._\text{function}) (\text{union} \text{ sigval})
\]

A pointer to the function to be notified.

\[
\text{pthread}._\text{attr} *\text{sigev}._\text{notify}._\text{attributes}
\]

A pointer to thread attributes. This must be NULL, or point to a structure initialized by \text{pthread}._\text{attr}._\text{init}() at the time of delivery.

\[
\text{void} *\text{sigev}._\text{value}.sival._\text{ptr}
\]

A value that’s to be passed to the notification function.

The initialization macro is:

\[
\text{SIGEVTHREAD}._\text{INIT} (\&\text{event}, \text{fn}, \text{value}, \text{attr})
\]

SIGEVUNBLOCK

Force a thread to become unblocked. No other fields in the structure are used.

The initialization macro is:

\[
\text{SIGEVUNBLOCK}._\text{INIT} (\&\text{event})
\]

Critical threads

If you’re using adaptive partitioning, you can use a \text{sigevent} to make a thread run as critical or not.
This feature was added in the QNX Neutrino Core OS 6.3.2. For more information, see the Adaptive Partitioning User’s Guide.

After setting up the `sigevent` structure as appropriate, use these macros to set or clear the hidden bit that makes a thread run as critical or not:

```
SIGEV_MAKE_CRITICAL(&event)

Make the targeted thread run as critical.
```

```
SIGEV_CLEAR_CRITICAL(&event)

Make the targeted thread not run as critical.
```

The receiving thread doesn’t have to do anything to make itself critical or noncritical; the adaptive partitioning scheduler does this automatically.

**CAUTION:** These macros use hidden bits in the `sigev_notify` member of the `sigevent` structure. Don’t compare this member directly to a value; use the `SIGEV_GET_TYPE()` macro instead, as described above.

**Classification:**

QNX Neutrino

**See also:**

d_s_create(), `InterruptAttach()`, `InterruptAttachEvent()`,
`iofunc_notify()`, `iofunc_notify_trigger()`, `ionotify()`, `lio_listio()`,
`mq_notify()`, `MsgDeliverEvent()`, `procmgr_event_notify()`, `pulse`,
`TimerCreate()`, `timer_create()`, `TimerInfo()`, `TimerTimeout()`,
`timer_timeout()`

“Neutrino IPC” in the Neutrino microkernel chapter of the System Architecture guide
**sigfillset()**

*Initialize a set to contain all signals*

**Synopsis:**

```c
#include <signal.h>

int sigfillset( sigset_t *set );
```

**Arguments:**

*set*  
A pointer to the `sigset_t` object that you want to initialize.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigfillset()` function initializes `set` to contain all signals.

**Returns:**

0  
Success.

-1  
An error occurred (`errno` is set).

**Examples:**

See `sigemptyset()`.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*continued...*
sigfillset()

© 2007, QNX Software Systems GmbH & Co. KG.

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`kill()`, `raise()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`,
`sigismember()`, `signal()`, `sigpending()`, `sigprocmask()`
**Synopsis:**

```c
#include <signal.h>

int sigismember( const sigset_t *set,  
                 int signo );
```

**Arguments:**

- **set** A pointer to the `sigset_t` object that you want to check.
- **signo** The signal that you want to check for membership in the set. For more information, see “POSIX signals” in the documentation for `SignalAction()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigismember()` function tests if `signo` is in the set pointed to by `set`.

**Returns:**

- 1 The `signo` is in the set.
- 0 The `signo` isn’t in the set.
- -1 An error occurred (`errno` is set).

**Errors:**

- `EINVAL` The signal `signo` isn’t valid.
Examples:

See `sigemptyset()`.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`kill()`, `raise()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`,
`sigfillset()`, `signal()`, `sigpending()`, `sigprocmask()`
siglongjmp()

Restore the environment saved by sigsetjmp(), including the signal mask

Synopsis:

```c
#include <setjmp.h>

void siglongjmp( sigjmp_buf env, int val );
```

Arguments:

- `env`: The environment saved by the most recent call to `sigsetjmp()`.
- `val`: The value that you want `sigsetjmp()` to return.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `siglongjmp()` function is a superset of the `longjmp()` function, but also restores the thread’s saved signal mask if (and only if) one was saved in the `env` argument by a previous call to `sigsetjmp()`.

WARNING: Don’t use `longjmp()` or `siglongjmp()` to restore an environment saved by a call to `setjmp()` or `sigsetjmp()` in another thread. If you’re lucky, your application will crash; if not, it’ll look as if it works for a while, until random scribbling on the stack causes it to crash.

Returns:

The same values that `longjmp()` returns.
Examples:

See `longjmp()`.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`longjmp()`, `sigaction()`, `sigprocmask()`, `sigsuspend()`
**Synopsis:**

```c
#include <unix.h>

#define sigmask(s) (1LL<<(s)-1))
```

**Arguments:**

- `s` The signal that you want to create a mask for. For more information, see “POSIX signals” in the documentation for `SignalAction()`.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

This macro constructs the mask for a given signal number. Use `sigmask()` in conjunction with `sigblock()`, `sigsetmask()`, and `sigunblock()`.

**Returns:**

The signal mask.

**Classification:**

- Unix

  **Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

  *continued...*
**sigmask()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

**See also:**

`kill()`, `sigaction()`, `sigblock()`, `signal()`, `sigpause()`, `sigsetmask()`, `sigunblock()`
signal()

Set handling for exceptional conditions

Synopsis:

```
#include <signal.h>

void ( * signal( int sig,
       void ( * func)(int) ) )( int );
```

Arguments:

- `sig`: The signal number (defined in `<signal.h>`). For more information, see “POSIX signals” in the documentation for `SignalAction()`.
- `func`: The function that you want to call when the signal is raised.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `signal()` function is used to specify an action to take place when certain conditions are detected while a program executes. See the `<signal.h>` header file for definitions of these conditions, and also refer to the System Architecture manual.

There are three types of actions that can be associated with a signal: SIG_DFL, SIG_IGN or a pointer to a function. Initially, all signals are set to SIG_DFL or SIG_IGN prior to entry of the `main()` routine. An action can be specified for each of the conditions, depending upon the value of the `func` argument, as discussed below.

**func is a function**

When `func` is a function name, that function is called in a manner equivalent to the following code sequence:

```c
/* "sig_no" is condition being signalled */
signal( sig_no, SIG_DFL );
(*func)( sig_no );
```
The `func` function may do the following:

- Return.
- Terminate the program by calling `exit()` or `abort()`.
- Call `longjmp()` or `siglongjmp()`. If you use `longjmp()` to return from a signal handler, the signal remains masked. You can use `siglongjmp()` to restore the mask to the state saved in a previous call to `sigsetjmp()`.

After returning from the signal-catching function, the receiving process resumes execution at the point at which it was interrupted.

The signal catching function is described as follows:

```c
void func( int sig_no )
{
    ... 
}
```

It isn’t possible to catch the SIGSTOP or SIGKILL signals.

Since signal-catching functions are invoked asynchronously with process execution, use the `atomic_ *`, `InterruptLock()`, and `InterruptUnlock()` functions for atomic operations.

**func is SIG_DFL**

If `func` is SIG_DFL, the default action for the condition is taken.

If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal is generated for its parent process, unless the parent process has set the `SA_NOCLDSTOP` flag (see `sigaction()`). While a process is stopped, any additional signals that are sent to the process aren’t delivered until the process is continued, except SIGKILL, which always terminates the receiving process.

Setting a signal action to SIG_DFL for a signal that is pending, and whose default action is to ignore the signal (for example, SIGCHLD), causes the pending signal to be discarded, whether or not it’s blocked.
func is SIG_IGN

If func is SIG_IGN, the indicated condition is ignored.

You can’t set the action for the SIGSTOP and SIGKILL signals to SIG_IGN.

Setting a signal action to SIG_IGN for a signal that’s pending causes the pending signal to be discarded, whether or not it is blocked.

If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is unspecified.

Handling a condition

When a condition is detected, it may be handled by a program, it may be ignored, or it may be handled by the usual default action (often causing an error message to be printed on the stderr stream followed by program termination).

A condition can be generated by a program using the raise() or kill() function

Returns:

The previous value of func for the indicated condition, or SIG_ERR if the request couldn’t be handled (errno is set to EINVAL).

Examples:

```c
#include <stdlib.h>
#include <signal.h>

sig_atomic_t signal_count;

void MyHandler( int sig_number )
{
    ++signal_count;
}

int main( void )
{
    signal( SIGFPE, MyHandler );    /* set own handler */
    signal( SIGABRT, SIG_DFL );     /* Default action */
    signal( SIGFPE, SIG_IGN );      /* Ignore condition */
    return (EXIT_SUCCESS);
}
```
Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

atomic_add(), atomic_add_value(), atomic_clr(),
atomic_clr_value(), atomic_set(), atomic_set_value(), atomic_sub(),
atomic_sub_value(), atomic_toggle(), atomic_toggle_value(),
InterruptLock(), InterruptUnlock(), kill(), longjmp(), raise(),
siglongjmp(), sigprocmask()
Examine and/or specify actions for signals

Synopsis:

```c
#include <sys/neutrino.h>

int SignalAction( pid_t pid,
                 void ( * sigstub ) (),
                 int signo,
                 const struct sigaction * act,
                 struct sigaction * oact );

int SignalAction_r( pid_t pid,
                    void * ( sigstub ) (),
                    int signo,
                    const struct sigaction * act,
                    struct sigaction * oact );
```

Arguments:

- `pid`: A process ID, or 0 for the current process.
- `sigstub`: The address of a signal stub handler. This is a small piece of code in the user’s space that interfaces the user’s signal handler to the kernel. The library provides a standard one, `__signalstub()`.
- `signo`: The signal whose action you want to set or get; see “POSIX signals,” below.
- `act`: NULL, or a pointer to a `sigaction` structure that specifies the new action for the signal. For more information, see “Signal actions,” below.
- `oact`: NULL, or a pointer to a `sigaction` structure where the function can store the old action.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**Description:**

The `SignalAction()` and `SignalAction_r()` kernel calls let the calling process examine or specify (or both) the action to be associated with a specific signal in the process `pid`. If `pid` is zero, the calling process is used. The argument `signo` specifies the signal.

You should call the POSIX `sigaction()` function or the ANSI `signal()` function instead of using these kernel calls directly.

These functions are identical except in the way they indicate errors. See the Returns section for details.

**POSIX signals**

The signals are defined in `<signal.h>`, and so are these global variables:

```c
char * const sys_siglist[]
    An array of signal names.
int sys_nsig    The number of entries in the `sys_siglist` array.
```

There are 32 POSIX 1003.1a signals, including:

- **SIGHUP**  Hangup.
- **SIGINT**  Interrupt.
- **SIGQUIT** Quit.
- **SIGILL**  Illegal instruction (not reset when caught).
- **SIGTRAP** Trace trap (not reset when caught).
- **SIGIOT**  IOT instruction.
- **SIGABRT** Used by `abort()`.
- **SIGEMT**  EMT instruction.
**SignalAction()**, **SignalAction_r()**

- **SIGFPE**: Floating point exception.
- **SIGKILL**: Kill (can’t be caught or ignored)
- **SIGBUS**: Bus error.
- **SIGSEGV**: Segmentation violation.
- **SIGSYS**: Bad argument to system call.
- **SIGPIPE**: Write on pipe with no reader.
- **SIGALRM**: Realtime alarm clock.
- **SIGTERM**: Software termination signal from kill.
- **SIGUSR1**: User-defined signal 1.
- **SIGUSR2**: User-defined signal 2.
- **SIGCHLD**: Death of child.
- **SIGPWR**: Power-fail restart.
- **SIGWINCH**: Window change.
- **SIGURG**: Urgent condition on I/O channel.
- **SIGPOLL**: System V name for SIGIO.
- **SIGIO**: Asynchronous I/O.
- **SIGSTOP**: Sendable stop signal not from tty.
- **SIGTSTP**: Stop signal from tty.
- **SIGCONT**: Continue a stopped process.
- **SIGTTIN**: Attempted background tty read.
- **SIGTTOU**: Attempted background tty write.
You can’t ignore or catch SIGKILL or SIGSTOP.

There are 16 POSIX 1003.1b realtime signals, including:

SIGRTMIN   First realtime signal.
SIGRTMAX   Last realtime signal.

The entire range of signals goes from _SIGMIN (1) to _SIGMAX (64).

Signal actions

If act isn’t NULL, then the specified signal is modified. If oact isn’t NULL, the previous action is stored in the structure it points to. You can use various combinations of act and oact to query or set (or both) the action for a signal.

The structure sigaction contains the following members:

void (*sa_handler)();
    The address of a signal handler or action for nonqueued signals.

void (*sa_sigaction) (int signo, siginfo_t *info, void *other);
    The address of a signal handler or action for queued signals.

sigset_t sa_mask
    An additional set of signals to be masked (blocked) during execution of the signal-catching function.

int sa_flags    Special flags that affect the behavior of the signal:
                • SA_NOCLDSTOP — don’t generate a SIGCHLD on the parent for children who stop via SIGSTOP. This flag is used only when the signal is SIGCHLD.
SignalAction(), SignalAction_r()

- SA_SIGINFO — queue this signal. The default is not to queue a signal delivered to a process. If a signal isn’t queued, and the same signal is set multiple times on a process or thread before it runs, only the last signal is delivered. If you set the SA_SIGINFO flag, the signals are queued, and they’re all delivered.

The sa_handler and sa_sigaction members of act are implemented as a union, and share common storage. They differ only in their prototype, with sa_handler being used for POSIX 1003.1a signals, and sa_sigaction being used for POSIX 1003.1b queued realtime signals. The values stored using either name can be one of:

**function**

- The address of a signal-catching function. See below for details.

**SIG_DFL**

- Use the default action for the signal:
  - SIGCHLD, SIGIO, SIGURG, and SIGWINCH — ignore the signal (SIG_IGN).
  - SIGSTOP — stop the process.
  - SIGCONT — continue the program.
  - All other signals — kill the process.

**SIG_IGN**

- Ignore the signal. Setting SIG_IGN for a signal that’s pending discards all pending signals, whether it’s blocked or not. New signals are discarded. If your process ignores SIGCHLD, its children won’t enter the zombie state and the process can’t use wait() or waitpid() to wait on their deaths.

The function member of sa_handler or sa_sigaction is always invoked with the following arguments:

**void handler(int signo, siginfo_t* info, void* other)**

If you have an old-style signal handler of the form:
void handler(int signo)

the microkernel passes the extra arguments, but the function simply ignores them.

While in the handler, signo is masked, preventing nested signals of the same type. In addition, any signals set in the sa_mask member of act are also ORed into the mask. When the handler returns through a normal return, the previous mask is restored, and any pending and now unmasked signals are acted on. You return to the point in the program where it was interrupted. If the thread was blocked in the kernel when the interruption occurred, the kernel call returns with an EINTR (see ChannelCreate() and SyncMutexLock() for exceptions to this).

When you specify a handler, you must provide the address of a signal stub handler for sigstub. This is a small piece of code in the user’s space that interfaces the user’s signal handler to the kernel. The library provides a standard one, ___signalstub().

The siginfo_t structure of the function in sa_handler or sa_sigaction contains at least the following members:

- **int si_signo**  The signal number, which should match the signo argument to the handler.

- **int si_code**  A signal code, provided by the generator of the signal:
  - SI_USER — the kill() function generated the signal.
  - SI_QUEUE — the sigqueue() function generated the signal.
  - SI_TIMER — a timer generated the signal.
  - SI_ASYNCIO — asynchronous I/O generated the signal.
  - SI_MESGQ — POSIX (not QNX) messages queues generated the signal.
union sigval si_value

A value associated with the signal, provided by the generator of the signal.

Signal handlers and actions are defined for the process and affect all threads in the process. For example, if one thread ignores a signal, then all threads ignore the signal.

You can target a signal at a thread, process or process group (see SignalKill()). When targeted at a process, at most one thread receives the signal. This thread must have the signal unblocked (see SignalProcmask()) to be a candidate for receiving it. All synchronously generated signals (e.g. SIGSEGV) are always delivered to the thread that caused them.

In a multithreaded process, if a signal terminates a thread, by default all threads and thus the process are terminated. You can override this standard POSIX behavior when you create the thread; see ThreadCreate().

CAUTION: If you use longjmp() to return from a signal handler, the signal remains masked. You can use siglongjmp() to restore the mask to the state saved by a previous call to sigsetjmp().

Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

 SignalAction() If an error occurs, -1 is returned and errno is set. Any other value returned indicates success.

 SignalAction_r() EOK is returned on success. This function does NOT set errno. If an error occurs, any value in the Errors section may be returned.
Errors:

- **EAGAIN**: The system was unable to allocate a signal handler. This indicated critically low memory.
- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided.
- **EINVAL**: The value of `signo` is less than 1 or greater than `_SIGMAX`, or you tried to set `SIGKILL` or `SIGSTOP` to something other than `SIG_DFL`.
- **EPERM**: The process doesn’t have permission to change the signal actions of the specified process.
- **ESRCH**: The process indicated by `pid` doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `abort()`, `ChannelCreate()`, `kill()`, `longjmp()`, `siglongjmp()`, `signal()`, `sigaction()`, `SignalKill()`, `SignalProcmask()`, `sigqueue()`, `sigsetjmp()`, `SyncMutexLock()`, `ThreadCreate()`, `wait()`, `waitpid()`
SignalKill(), SignalKill_r()

Send a signal to a process group, process, or thread

Synopsis:

```c
#include <sys/neutrino.h>

int SignalKill( uint32_t nd,
                pid_t pid,
                int tid,
                int signo,
                int code,
                int value );

int SignalKill_r( uint32_t nd,
                   pid_t pid,
                   int tid,
                   int signo,
                   int code,
                   int value );
```

Arguments:

- **nd**  The node descriptor of the node on which to look for *pid* and *tid*. To search the local node, set *nd* to ND_LOCAL_NODE or 0.
- **pid**  0, or the ID of the process to send the signal to; see below.
- **tid**  0, or the ID of the thread to send the signal to; see below.
- **signo**  The signal that you want to send. There are a total of 64 signals available. Of these, at least 8 are POSIX realtime signals that range from SIGRTMIN to SIGRTMAX. For a complete list of signals, see “POSIX signals” in the documentation for SignalAction(). Valid user signals range from 1 to (NSIG - 1).
- **code, value**  The code and value associated with the signal; see SignalAction().
Library:

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

Description:

The \texttt{SignalKill()} and \texttt{SignalKill\_r()} kernel calls send the signal \texttt{signo} with a code specified by \texttt{code} and a value specified by \texttt{value} to a process group, process, or thread.

These functions are identical except in the way they indicate errors. See the Returns section for details.

If \texttt{signo} is zero, no signal is sent, but the validity of \texttt{pid} and \texttt{tid} are checked. You can use this as a test for existence.

\texttt{SignalKill()} implements the capabilities of the POSIX functions \texttt{kill()}, \texttt{sigqueue()}, and \texttt{pthread\_kill()} in one call. Consider using these functions instead of invoking these kernel calls directly.

The \texttt{pid} and \texttt{tid} determine the target of the signal, as follows:

<table>
<thead>
<tr>
<th>\texttt{pid}</th>
<th>\texttt{tid}</th>
<th>\texttt{target}</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 0</td>
<td></td>
<td>Hit the process group of the caller</td>
</tr>
<tr>
<td>&lt; 0</td>
<td></td>
<td>Hit a process group identified by \texttt{-pid}</td>
</tr>
<tr>
<td>&gt; 0 = 0</td>
<td></td>
<td>Hit a single process identified by \texttt{pid}</td>
</tr>
<tr>
<td>&gt; 0 &gt; 0</td>
<td></td>
<td>Hit a single thread in process \texttt{pid} identified by \texttt{tid}</td>
</tr>
</tbody>
</table>

If the target is a thread, the signal is always delivered to exactly that thread. If the thread has the signal blocked — see \texttt{SignalProcmask()} — the signal remains pending on the thread.

If the target is a process, the signal is delivered to a thread that has the signal unblocked; see \texttt{SignalProcmask()}, \texttt{SignalSuspend()}, and \texttt{SignalWaitinfo()}. If multiple threads have the signal unblocked, only
one thread is given the signal. Which thread receives the signal isn’t
deterministic. To make it deterministic, you can:

- Have all threads except one block all signals; that thread handles
  all signals.
  
  Or:

- Target the signals to specific threads.

If all threads have the signal blocked, it’s made pending on the
process. The first thread to unblock the signal receives the pending
signal. If a signal is pending on a thread, it’s never retargetted to the
process or another thread, regardless of changes to the signal-blocked
mask.

If the target is a process group, the signal is delivered as above to each
process in the group.

A multithreaded application typically has one thread responsible for
catching most or all signals. Threads that don’t wish to be directly
involved with signals block all signals in their mask.

The signal-blocked mask is maintained on a per-thread basis. The
signal-ignore mask and signal handlers are maintained at the process
level and are shared by all threads.

If multiple signals are delivered before the target can run and process
the signals, the system queues them in priority order if the
SA_SIGINFO bit was set for signo. Lower numbered signals have
greater priority. If the SA_SIGINFO bit isn’t set for signo, then at most
one signal is queued at any time. Additional signals with the same
signo replace existing ones. This is the default behavior for POSIX
signal handlers installed using the old signal() function. The newer
sigaction() function lets you control queuing or not on a per-signal
basis. Signals with a code of SI_TIMER are never queued.

The code and value are always saved with the signal. This allows you
to deliver data with the signal whether or not SA_SIGINFO has been
set on the signo. If SA_SIGINFO is set, you can use signals to deliver
small amounts of data without loss. If you wish to pass significant
data, you may wish to consider using `MsgSendPulse()` and `MsgSendv()`, which deliver data with much greater efficiency. When a thread receives a signal by a signal handler or `SignalWaitinfo()` call, it can retrieve the `signo`, `code` and `value` from a `siginfo_t` structure, which contains at least the following members:

```c
int si_signo   The signal number.
int si_code    The signal code.
union sigval si_value
               The signal value.
```

The value of `si_code` is limited to an 8-bit signed value as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-128 &lt;= si_code &lt;= 0</td>
<td>User values</td>
</tr>
<tr>
<td>0 &lt; signo &lt;= 127</td>
<td>System values generated by the kernel</td>
</tr>
</tbody>
</table>

Some of the common user values defined by POSIX are:

- `SI_USER` — the `kill()` function generated the signal.
- `SI_QUEUE` — the `sigqueue()` function generated the signal.
- `SI_TIMER` — a timer generated the signal.
- `SI_ASYNCIO` — asynchronous I/O generated the signal.
- `SI_MESGQ` — POSIX (not QNX) messages queues generated the signal.

A successful return from this function means the signal has been delivered. What the process(es) or thread does with the signal isn’t considered.

If a thread delivers signals that the receiving process has marked as queued faster than the receiver can consume them, the kernel may fail
the call if it runs out of signal queue entries. If the `signo`, `code`, and `value` don’t change, the kernel performs signal compression by saving an 8-bit count with each queued signal.

**Blocking states**

None. In the network case, lower priority threads may run.

**Returns:**

The only difference between these functions is the way they indicate errors:

- `SignalKill()` If an error occurs, -1 is returned and sets `errno`. Any other value returned indicates success.
- `SignalKill_r()` EOK is returned on success. This function does **not** set `errno`. If an error occurs, any value in the Errors section may be returned.

**Errors:**

- **EINV AL** The value of `signo` is less than 0 or greater than (_NSIG -1).
- **ESRCH** The process or process group indicated by `pid` or thread indicated by `tid` doesn’t exist.
- **EPERM** The process doesn’t have permission to send the signal to any receiving process.
- **EAGAIN** The kernel had insufficient resources to enqueue the signal.

**Classification:**

QNX Neutrino
**SignalKill(), SignalKill_r()**

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

pthread_kill(), kill(), SignalAction(), SignalProcmask(), SignalSuspend(), SignalWaitinfo(), sigqueue()
SignalProcmask(), SignalProcmask_r()

Modify or examine the signal-blocked mask of a thread

Synopsis:

```c
#include <sys/neutrino.h>

int SignalProcmask( pid_t pid,
                     int tid,
                     int how,
                     const sigset_t* set,
                     sigset_t* oldset );

int SignalProcmask_r( pid_t pid,
                       int tid,
                       int how,
                       const sigset_t* set,
                       sigset_t* oldset );
```

Arguments:

- `pid` 0, or a process ID; see below.
- `tid` 0, or a thread ID; see below.
- `how` The manner in which you want to change the set:
  - SIG_BLOCK — the resulting set is the union of the current set and the signal set pointed to by `set`.
  - SIG_UNBLOCK — the resulting set is the intersection of the current set and the signal set pointed to by `set`.
  - SIG_SETMASK — the resulting set is the signal set pointed to by `set`.

As a special case, you can use the `how` argument to query the current set of pending signals:

- SIG_PENDING — the combined set of pending signals on the thread and process are saved in the signal set pointed to by `oldset`. The `set` argument is ignored.

- `set` NULL, or a pointer to a `sigset_t` object that specified the set of signals to be used to change the currently blocked set.
**oldset**

NULL, or a pointer to a `sigset_t` object where the function can store the previous blocked mask.

You can use various combinations of `set` and `oldset` to query or change (or both) the signal-blocked mask for a signal.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

These kernel calls modify or examine the signal-blocked mask of the thread `tid` in process `pid`. If `pid` is zero, the current process is assumed. If `tid` is zero, `pid` is ignored and the calling thread is used.

The `SignalProcmask()` and `SignalProcmask_r()` functions are identical, except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `pthread_sigmask()` or `sigprocmask()`.

When a signal is unmasked, the kernel checks for pending signals on the thread and, if there aren’t any pending, checks for pending signals on the process:

<table>
<thead>
<tr>
<th>Check</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal pending on thread</td>
<td>The signal is immediately acted upon.</td>
</tr>
<tr>
<td>Signal pending on process</td>
<td>The signal is moved to the thread and is immediately acted upon.</td>
</tr>
</tbody>
</table>

*continued...*
Check | Action
--- | ---
No signal pending | No signal action performed until delivery of an unblocked signal.

It isn’t possible to block the SIGKILL or SIGSTOP signals.

When a signal handler is invoked, the signal responsible is automatically masked before its handler is called; see `SignalAction()`.

If the handler returns normally, the operating system restores the signal mask present just before the handler was called as an atomic operation. Changes made using `SignalProcmask()` in the handler are undone.

When a signal is targeted at a process, the kernel delivers it to at most one thread (see `SignalKill()` that has the signal unblocked. If multiple threads have the signal unblocked, only one thread is given the signal. Which thread receives the signal isn’t deterministic. To make it deterministic, you can:

- Have all threads except one block all signals; that thread handles all signals.

  Or:

- Target signals to specific threads.

If all threads have the signal blocked, it’s made pending on the process. The first thread to unblock the signal receives the pending signal. If a signal is pending on a thread, it’s never retargetted to the process or another thread, regardless of changes to the signal-blocked mask.

Signals targeted at a thread always affect that thread alone.

**Blocking states**

These calls don’t block.
Returns:

The only difference between these functions is the way they indicate errors:

SignalProcmask()

If an error occurs, -1 is returned and errno is set. Any other value returned indicates success.

SignalProcmask_r()

EOK is returned on success. This function does NOT set errno. If an error occurs, any value in the Errors section may be returned.

Errors:

EAGAIN The system was unable to allocate a signal handler. This indicates critically low memory.

EFAULT A fault occurred when the kernel tried to access the buffers provided.

EINVAL The value of how is invalid, or you tried to set SIGKILL or SIGSTOP to something other than SIG_DFL.

EPERM The process doesn’t have permission to change the signal mask of the specified process.

ESRCH The process indicated by pid or thread indicated by tid doesn’t exist.

Classification:

QNX Neutrino

Safety

Cancellation point      No

continued…
SignalProcmask(), SignalProcmask_r()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_setmask(), SignalAction(), SignalKill(), sigprocmask()
**SignalSuspend(), SignalSuspend_r()**

Suspend a thread until a signal is received

**Synopsis:**

```
#include <sys/neutrino.h>

int SignalSuspend( const sigset_t* set );

int SignalSuspend_r( const sigset_t* set );
```

**Arguments:**

- `set` A pointer to a `sigset_t` object that specifies the signals you want to wait for.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

These kernel calls replace the thread’s signal mask with the set of signals pointed to by `set` and then suspends the thread until delivery of a signal whose action is either to execute a signal-catching function (then return), or to terminate the thread. On return, the previous signal mask is restored.

The `SignalSuspend()` and `SignalSuspend_r()` functions are identical, except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `sigsuspend()`.

Attempts to block SIGKILL or SIGSTOP are ignored. This is done without causing an error.

If you’re using `SignalSuspend()` to synchronously wait for a signal, consider using the more efficient POSIX 1003.1b realtime `sigwaitinfo()` call.
**SignalSuspend(), SignalSuspend_r()**

**Blocking states**

STATE_SIGSUSPEND

The calling thread blocks waiting for a signal.

**Returns:**

The only difference between these functions is the way they indicate errors.

Since `SignalSuspend()` and `SignalSuspend_r()` block until interrupted, there’s no successful return value.

- `SignalSuspend()`
  - `-1` is always returned and `errno` is set.

- `SignalSuspend_r()`
  - `errno` is **NOT** set, a value in the Errors section is returned.

If the signal handler calls `longjmp()` or `siglongjmp()`, `SignalSuspend()` and `SignalSuspend_r()` don’t return.

**Errors:**

- **EINTR** The call was interrupted by a signal (this is the normal error).
- **EFAULT** A fault occurred when the kernel tried to access the buffers provided.
- **ETIMEDOUT** A kernel timeout unblocked the call. See `TimerTimeout()`.

**Classification:**

QNX Neutrino
**SignalSuspend(), SignalSuspend_r()**

© 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*SignalKill(), sigsuspend()*
Synopsis:

```c
#include <sys/neutrino.h>

int SignalWaitinfo( const sigset_t* set,
                    siginfo_t* info );

int SignalWaitinfo_r( const sigset_t* set,
                      siginfo_t* info );
```

Arguments:

- `set` A pointer to a `sigset_t` object that specifies the signals you want to wait for.
- `info` NULL, or a pointer to a `siginfo_t` structure where the function can store information about the signal.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SignalWaitinfo()` and `SignalWaitinfo_r()` kernel calls select the pending signal from the set specified by `set`. If no signal in `set` is pending at the time of the call, the thread blocks until one or more signals in `set` become pending or until interrupted by an unblocked, caught signal.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `sigwaitinfo()`.

If the `info` argument isn’t NULL, information on the selected signal is stored there as follows:
**SignalWaitinfo(), SignalWaitinfo_r()** © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th><code>siginfo_t</code> member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>si_signo</code></td>
<td>Selected signal number</td>
</tr>
<tr>
<td><code>si_code</code></td>
<td>Signal code</td>
</tr>
<tr>
<td><code>si_value</code></td>
<td>Signal value</td>
</tr>
</tbody>
</table>

If, while `SignalWaitinfo()` is waiting, a caught signal occurs that isn’t blocked, the signal handler is invoked and `SignalWaitinfo()` is interrupted with an error of EINTR.

**Blocking states**

STATE_SIGWAITINFO

The calling thread blocks waiting for a signal.

**Returns:**

The only difference between these functions is the way they indicate errors:

`SignalWaitinfo()`

A signal number. If an error occurs, -1 is returned and `errno` is set.

`SignalWaitinfo_r()`

A signal number. This function does NOT set `errno`. If an error occurs, the negative of a value from the Errors section is returned.

**Errors:**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINTR</td>
<td>The call was interrupted by a signal.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the buffers provided.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>A kernel timeout unblocked the call. See <code>TimerTimeout()</code>.</td>
</tr>
</tbody>
</table>
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

SignalKill(), SignalKill_r(), sigwaitinfo()
**significand(), significandf()** © 2007, QNX Software Systems GmbH & Co. KG.

*Compute the “significant bits” of a floating-point number*

**Synopsis:**

```
#include <math.h>

double significand ( double x );

float significandf ( float x );
```

**Arguments:**

- \( x \) A floating-point number.

**Library:**

```
libm
```

Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `significand()` and `significandf()` functions are math functions that compute the “significant bits” of a floating-point number.

When encoding a floating-point number into binary notation, you remove the sign, and then shift the bits to the right or left until the shifted result is in the range \([0.5, 1)\). The negative of the number of positions shifted is the **exponent** of the number, and the shifted result is the **significand**.

If \( x \) equals \( \text{sig} \times 2^n \) with \( 1 < \text{sig} < 2 \), then `significand(x)` returns `sig` for exercising the fraction-part(F) test vector. The function `significand(x)` isn’t defined when \( x \) is one of:

- 0
- positive or negative infinity
- NAN.
Returns:

\[ \text{scalb} \left( x, \text{(double)} \cdot \text{-ilogb} \left( x \right) \right) \]

Since \( \text{significand}(x) = \text{scalb}(x, -\text{ilogb}(x)) \) where \( \text{ilogb}() \) returns the exponent part of \( x \) and \( \text{scalb}(x, n) \) returns \( a \), such that \( x = a \times 2^B \), then:

\[
\begin{array}{ll}
\text{When } x \text{ is: } & \text{scalbn}(x, n) \text{ returns:} \\
\pm \infty & x \\
\text{NAN} & \text{NAN}
\end{array}
\]

Examples:

```c
#include <stdio.h>
#include <errno.h>
#include <inttypes.h>
#include <math.h>
#include <fpstatus.h>

int main(int argc, char** argv)
{
    double a, b, c, d;
    a = 5;
    b = ilogb(d);
    printf("The exponent part of \%f is \%f \n", a, b);
    c = significand(a);
    printf("%f = %f \times (2 ^ %f) \n", a, c, b);
    return(0);
}
```

produces the output:

The exponent part of 5.000000 is -895.000000
5.000000 = 1.250000 \times (2 ^ -895.000000)

Classification:

Unix
**significand(), significandf()**

### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*ilogb(), scalb(), scalbn()*
Sigpause() -- Wait for a signal

Synopsis:

```c
#include <signal.h>

int sigpause( int sig );
```

Arguments:

- `sig` A mask containing the signal number that you want to wait for.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigpause()` function assigns `sig` to the set of masked signals and then waits for a signal to arrive; on return, the set of masked signals is restored. The `mask` argument is usually 0 to indicate that no signals are now to be blocked. This function always terminates by being interrupted, returning -1, and setting `errno` to EINTR.

In normal usage, a signal is blocked using `sigblock()`. To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using `sigpause()` with the mask returned by `sigblock()`.

It isn’t possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

Returns:

-1; `errno` is set to EINTR.
**sigpause()**

**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

**See also:**

`kill()`, `sigaction()`, `sigblock()`, `sigmask()`, `signal()`, `sigsetmask()`, `sigsuspend()`, `sigunblock()`
Examine the set of pending, masked signals for a process

**Synopsis:**

```c
#include <signal.h>

int sigpending( sigset_t *set );
```

**Arguments:**

- `set` A pointer to a `sigset_t` object that the function sets to indicate the pending, masked signals.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigpending()` function is used to examine the set of pending signals that are masked (blocked) from delivery to the calling thread and that are pending on the calling process or thread. They’re saved in the signal set pointed to by `set`.

**Returns:**

- `0` Success.
- `-1` An error occurred (`errno` is set).

**Errors:**

- `EFAULT` A fault occurred while accessing the buffer pointed to by `set`.

**Examples:**

See `sigprocmask()`.
Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

kill(), raise(), sigaction(), sigaddset(), sigdelset(), sigemptyset(),
sigfillset(), sigismember(), signal(), sigprocmask()
**Synopsis:**

```c
#include <signal.h>

int sigprocmask( int how,
    const sigset_t *set,
    sigset_t *oset );
```

**Arguments:**

- `how` The manner in which you want to change the set:
  - `SIG_BLOCK` — add the signals pointed to by `set` to the thread mask.
  - `SIG_UNBLOCK` — remove the signals pointed to by `set` from the thread mask.
  - `SIG_SETMASK` — set the thread mask to be the signals pointed to by `set`.

- `set` NULL, or a pointer to a `sigset_t` object that defines the signals that you want to change in the thread’s signal mask. If this argument is NULL, the `how` argument is ignored.

- `oset` NULL, or a pointer to a `sigset_t` object that the function sets to indicate the thread’s current signal mask.

**Library:**

-libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigprocmask()` function is used to examine or change (or both) the signal mask for the calling thread. If the value of `set` isn’t NULL, it points to a set of signals to be used to change the currently blocked set.
The set argument isn’t changed. The resulting set is maintained in the process table of the calling thread. If a signal occurs on a signal that’s masked, it becomes pending, but doesn’t affect the execution of the process. You can examine pending signals by calling sigpending(). When a pending signal is unmasked, it’s acted upon immediately, before this function returns.

When a signal handler is invoked, the signal responsible is automatically masked before its handler is called. If the handler returns normally, the operating system restores the signal mask present just before the handler was called as an atomic operation. Changes made using sigprocmask() in the handler are undone.

The sigaction() function lets you specify any mask that’s applied before a handler is invoked. This can simplify multiple signal handler design.

**Returns:**

0 Success.

-1 An error occurred (errno is set).

**Errors:**

EAGAIN Insufficient system resources are available to mask the signals.

EFAULT A fault occurred trying to access the buffers provided.

EINVAL The value of how is invalid, or you tried to set SIGKILL or SIGSTOP to something other than SIG_DFL.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main( void )
{
    sigset_t set, oset, pset;
    ...
```
sigemptyset( &set );
sigaddset( &set, SIGINT );
sigprocmask( SIG_BLOCK, &set, &oset );
printf( "Old set was %8.8ld,\n", oset );

sigpending( &pset );
printf( "Pending set is %8.8ld,\n", pset );

kill( getpid(), SIGINT );
sigpending( &pset );
printf( "Pending set is %8.8ld,\n", pset );
sigprocmask( SIG_UNBLOCK, &set, &oset );

/* The program terminates with a SIGINT */
return( EXIT_SUCCESS );
}

Classification:

POSIX 1003.1 THR

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

kill(), raise(), sigaction(), sigaddset(), sigdelset(), sigemptyset(),
sigfillset(), sigismember(), signal(), SignalProcmask(), sigpending()
sigqueue()

Queue a signal to a process

Synopsis:

```c
#include <signal.h>

int sigqueue ( pid_t pid,
               int signo,
               const union sigval value );
```

Arguments:

- `pid` The ID of the process that you want to signal.
- `signo` Zero, or the number of the signal that you want to queue for the process. For more information, see “POSIX signals” in the documentation for SignalAction().
- `value` The value to queue with the signal.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigqueue()` function causes the signal, `signo` to be sent with the specified value to the process, `pid`. If `signo` is zero, error checking is performed, but no signal is actually sent. This is one way of checking to see if `pid` is valid.

The condition required for a process to have permission to queue a signal to another process is the same as for the `kill()` function — the real or effective user ID of the sending process must match the real or effective user ID of the receiving process.

The `sigqueue()` function returns immediately. If SA_SIGINFO is set for `signo` and if the resources are available to queue the signal, the signal is queued and sent to the receiving process. If SA_SIGINFO isn’t set for the `signo`, then `signo` is sent to the receiving process if the signal isn’t already pending.
If `pid` causes `signo` to be generated for the sending process, and if `signo` isn’t blocked for the calling thread and if no other thread has `signo` unblocked or is waiting in a `sigwait()` function for `signo`, then either `signo` or at least one pending unblocked signal is delivered to the calling thread before `sigqueue()` returns.

Should any of multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected for delivery, the lowest numbered one is delivered. The selection order between realtime and nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified.

**Returns:**

0  Success.

-1  An error occurred; `errno` is set.

**Errors:**

- **EAGAIN**  No resources were available to queue the signal. The process has already queued the maximum number of signals as returned by:

```c
sysconf(_SC_SIGQUEUE_MAX)
```

that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.

- **EINVAL**  The value of the `signo` argument is an invalid or unsupported signal number.

- **ENOSYS**  The function `sigqueue()` isn’t supported by this implementation.

- **EPERM**  The process doesn’t have the appropriate privilege to send the signal to the receiving process.

- **ESRCH**  The process `pid` doesn’t exist.
Classification:

POSIX 1003.1 RTS

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`kill()`, `signal()`
sigsetjmp()

Save the environment, including the signal mask

Synopsis:

```c
#include <setjmp.h>

int sigsetjmp( sigjmp_buf env,
               int savemask );
```

Arguments:

- `env` A buffer where the function can save the calling environment.
- `savemask` Nonzero if you want to save the process’s current signal mask, otherwise 0.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigsetjmp()` function behaves in the same way as the `setjmp()` function when `savemask` is zero. If `savemask` is nonzero, then `sigsetjmp()` also saves the thread’s current signal mask as part of the calling environment.

**WARNING:** Don’t use `longjmp()` or `siglongjmp()` to restore an environment saved by a call to `setjmp()` or `sigsetjmp()` in another thread. If you’re lucky, your application will crash; if not, it’ll look as if it works for a while, until random scribbling on the stack causes it to crash.

Returns:

Zero on the first call, or nonzero if the return is the result of a call to `siglongjmp()`.
sigsetjmp() © 2007, QNX Software Systems GmbH & Co. KG.

Examples:

See setjmp().

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

setjmp(), sigaction(), sigprocmask(), sigsuspend()
**sigsetmask()**

Set the mask of signals to block

**Synopsis:**
```
#include <unix.h>

int sigsetmask(int mask);
```

**Arguments:**

*mask*  
A bitmask of the signals that you want to block.

**Library:**

*libc*  
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigsetmask()` function sets the current signal mask (those signals that are blocked from delivery). Signals are blocked if the corresponding bit in `mask` is a 1; the macro `sigmask()` is provided to construct the mask for a given signum.

In normal usage, a signal is blocked using `sigblock()`. To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using `sigpause()` with the mask returned by `sigblock()`.

It isn’t possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

**Returns:**

The previous set of masked signals.

**Classification:**

Unix
### sigsetmask()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

**See also:**

- `kill()`, `sigaction()`, `sigblock()`, `sigmask()`, `signal()`, `sigpause()`, `sigprocmask()`, `sigunblock()`
sigsuspend()

Replace the signal mask, and then suspend the thread

Synopsis:

```c
#include <signal.h>

int sigsuspend( const sigset_t *sigmask );
```

Arguments:

- `sigmask`: A pointer to a `sigset_t` object that specifies the signals that you want in the thread’s signal mask.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigsuspend()` function replaces the thread’s signal mask with the set of signals pointed to by `sigmask` and then suspends the thread until delivery of a signal whose action is either to execute a signal-catching function (then return), or to terminate the thread.

Returns:

-1 (if the function returns); `errno` is set.

Errors:

- `EFAULT`: A fault occurred trying to access the buffers provided.
- `EINTR`: A signal was caught by the calling thread, and control is returned from the signal-catching function.

Examples:

```c
/*
 * This program pauses until a signal other than
 * a SIGINT occurs. In this case a SIGALRM.
 */
```
```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

sigset_t set;

int main( void )
{
    sigemptyset( &set );
    sigaddset( &set, SIGINT );

    printf( "Program suspended and immune to breaks.\n" );
    printf( "A SIGALRM will terminate the program" );
    printf( " in 10 seconds.\n" );
    alarm( 10 );
    sigsuspend( &set );
    return( EXIT_SUCCESS );
}
```

**Classification:**

POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pause()`, `sigaction()`, `sigpending()`, `sigprocmask()`
sigtimedwait() — Wait for a signal or a timeout

Synopsis:

```c
#include <signal.h>

int sigtimedwait( const sigset_t *set,
                  siginfo_t *info,
                  const struct timespec *timeout );
```

Arguments:

- **set**: A set of signals from which the function selects a pending signal.
- **info**: If this argument is NULL, the selected signal is returned by `sigwaitinfo()`; otherwise, the selected signal is stored in the `si_signo` member of `info`, and the cause of the signal is stored in the `si_code` member.
- **timeout**: NULL, or a pointer to a `timespec` structure that specifies the maximum time to wait for a pending signal.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sigtimedwait()` function selects a pending signal from `set`, atomically clears it from the set of pending signals in the process, and returns that signal number.

If any value is queued to the selected signal, the first queued value is dequeued and, if the `info` argument is non-NULL, the value is stored in the `si_value` member of `info`. The system resources used to queue the signal are released and made available to queue other signals. If no value is queued, the content of the `si_value` member is undefined.

If no further signals are queued for the selected signal, the pending indication for that signal is reset.
If none of the signals specified by set are pending, sigtimedwait() waits for the time interval specified by the timespec structure timeout. If timeout is zero and if none of the signals specified by set are pending, then sigtimedwait() returns immediately with an error. If timeout is NULL, sigtimedwait() behaves the same as sigwaitinfo().

**Returns:**

The selected signal number, or -1 if an error occurred (errno is set).

**Errors:**

- **EAGAIN**: The timeout expired before a signal specified in set was generated, or all kernel timers are in use.
- **EFAULT**: A fault occurred while accessing the provided buffers.
- **EINTR**: The wait was interrupted by an unblocked, caught signal.
- **EINVAL**: The timeout argument specified a tv_nsec value less than zero or greater than or equal to 1000 million or set contains an invalid or unsupported signal number.

**Classification:**

POSIX 1003.1 RTS

**Safety**

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
see also:

`pause()`, `pthread_sigmask()`, `sigaction()`, `SignalWaitinfo()`,
`sigpending()`, `sigsuspend()`, `sigwaitinfo()`, `timespec`
**sigunblock()**

Unblock signals

**Synopsis:**

```c
#include <unix.h>

int sigunblock( int mask );
```

**Arguments:**

`mask`  A bitmask of the signals that you want to unblock.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigunblock()` function removes the signals specified in `mask` from the set of signals currently being blocked from delivery. Signals are unblocked if the appropriate bit in `mask` is a 1; the macro `sigmask()` is provided to construct the mask for a given signum. The `sigunblock()` returns the previous mask. You can restore the previous mask by calling `sigsetmask()`.

In normal usage, a signal is blocked using `sigblock()`. To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using `sigpause()` with the mask returned by `sigblock()`.

It isn’t possible to block SIGKILL, SIGSTOP, or SIGCONT; this restriction is silently imposed by the system.

**Returns:**

The previous set of masked signals.
sigunblock()

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multithreaded applications is unsupported.

See also:

kill(), sigaction(), sigblock(), sigmask(), signal(), sigpause(), sigprocmask(), sigsetmask()
**sigwait()**

*Wait for a pending signal*

**Synopsis:**
```
#include <signal.h>

int sigwait( const sigset_t *set,  
             int *sig );
```

**Arguments:**
- `set` A pointer to a `sigset_t` object that specifies the signals you want to wait for.
- `sig` A pointer to a location where the function can store the signal that it cleared.

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigwait()` function selects a pending signal from `set`, atomically clears it from the set of pending signals in the system, and returns that signal number in `sig`. If there are multiple signals queued for the signal number selected, the first signal causes a return from `sigwait()` and the rest remain queued. If no signal in `set` is pending at the time of the call, the thread is suspended until one or more becomes pending.

The signals defined by `set` must be blocked before you call `sigwait()`, otherwise, the behavior is undefined. The effect of `sigwait()` on the signal actions for the signals in `set` is unspecified.

If more than one thread is using `sigwait()` to wait for the same signal, only one of the threads returns from `sigwait()` with the signal number — which one is unspecified.
Returns:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success (that is, one of the signals specified by set is pending or has been generated).</td>
</tr>
<tr>
<td>EINTR</td>
<td>The <code>sigwait()</code> function was interrupted by a signal.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>set</code> argument contains an invalid or unsupported signal number.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred while accessing the provided buffers.</td>
</tr>
</tbody>
</table>

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pause()`, `pthread_sigmask()`, `sigaction()`, `sigpending()`, `sigsuspend()`
**sigwaitinfo()**

Wait for a specified signal and return its information

**Synopsis:**

```c
#include <signal.h>

int sigwaitinfo ( const sigset_t *set,
                  siginfo_t *info );
```

**Arguments:**

- `set` A pointer to a `sigset_t` object that specifies the signals you want to wait for.
- `info` NULL, or a pointer to a `siginfo_t` structure where the function can store information about the signal.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sigwaitinfo()` function selects a pending signal from `set`, atomically clears it from the set of pending signals in the system, and returns that signal number.

If there’s no pending signal, `sigwaitinfo()` blocks until the specified signal is pending. If the specified signal is already pending, upon call to `sigwaitinfo()`, the call immediately returns without blocking.

If `info` is NULL, the selected signal is returned by `sigwaitinfo()`, otherwise, the selected signal is stored in the `si_signo` member of `info` and the cause of the signal is stored in the `si_code` member.

If any value is queued to the selected signal, the first queued value is dequeued and, if the `info` argument is non-NULL, the value is stored in the `si_value` member of `info`. The system resources used to queue the signal are released and made available to queue other signals. If no value is queued, the content of the `si_value` member is undefined.
sigwaitinfo()

If no further signals are queued for the selected signal, the pending indication for that signal is reset.

Returns:

A signal number, or -1 if an error occurred (errno is set).

Errors:

EFAULT A fault occurred while accessing the buffers.
EINTR The wait was interrupted by an unblocked, caught signal.

Classification:

POSIX 1003.1 RTS

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pause(), pthread_sigmask(), sigaction(), SignalWaitinfo(), sigpending(), sigsuspend(), sigtimedwait(), sigwait()


**sin(), sinf()**

*Calculate the sine of an angle*

**Synopsis:**

```c
#include <math.h>

double sin( double x );

float sinf( float x );
```

**Arguments:**

`x` The angle, in radians, for which you want to compute the sine.

**Library:**

`libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**

The `sin()` and `sinf()` functions compute the sine (specified in radians) of `x`. An argument with a large magnitude may yield a result with little or no significance.

**Returns:**

The sine value.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%f\n", sin(.5) );
}
```

---

© 2007, QNX Software Systems GmbH & Co. KG.
return( EXIT_SUCCESS );
}

produces the output:

0.479426

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

acos(), asin(), atan(), atan2(), cos(), tan()
Compute the hyperbolic sine

Synopsis:

```c
#include <math.h>

double sinh( double x );

float sinhf( float x );
```

Arguments:

- `x` The angle, in radians, for which you want to compute the hyperbolic sine.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

The `sinh()` and `sinhf()` functions compute the hyperbolic sine (specified in radians) of `x`. A range error occurs if the magnitude of `x` is too large.

Returns:

The hyperbolic sine value.

Note:

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
```

3042 C Library — S September 10, 2007
```c
    printf( "%f\n", sinh(.5) );
    return( EXIT_SUCCESS );
}
```

produces the output:

```
0.521095
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`cosh()`, `errno`, `tanh()`
sleep()
Suspend a thread for a given length of time

Synopsis:
#include <unistd.h>

unsigned int sleep( unsigned int seconds );

Arguments:
seconds The number of realtime seconds that you want to suspend the thread for.

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The sleep() function suspends the calling thread until the number of realtime seconds specified by the seconds argument have elapsed, or the thread receives a signal whose action is either to terminate the process or to call a signal handler. The suspension time may be greater than the requested amount, due to the scheduling of other, higher priority threads by the system.

Returns:
0 if the full time specified was completed; otherwise, the number of seconds unslept if interrupted by a signal.

Errors:
EAGAIN No timer resources were available to satisfy the request.

Examples:
/*
 * The following program sleeps for the number of seconds specified in argv[1].
 */
```
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    unsigned seconds;
    seconds = (unsigned) strtol( argv[1], NULL, 0 );
    sleep( seconds );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

alarm(), delay(), errno, nanosleep(), timer_create(), timer_gettime(),
timer_settime(), usleep()
_sleepon_broadcast()

Synopsis:

```
#include <pthread.h>

int _sleepon_broadcast( sleepon_t * l,
    const volatile void * addr );
```

Arguments:

- `l`: A pointer to a `sleepon_t` that you created by calling `_sleepon_init()`.
- `addr`: The handle that the threads are waiting on. The value of `addr` is typically a data structure that controls a resource.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_signal()` and `_sleepon_broadcast()` functions are very similar:

- `_sleepon_signal()` wakes up a single thread that’s waiting on the key, `addr`.
- `_sleepon_broadcast()` wakes up all threads that are waiting on the key, `addr`.

The waiting threads must have used the same sleepon, `l` and key, `addr`, in order to be woken up.

To be woken up, the calling threads must have been locked by `_sleepon_lock()`.
Returns:

0  Success.
≠0  Failure.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

 pthread_sleepon_broadcast(), pthread_sleepon_signal(),
     _sleepon_destroy(), _sleepon_init(), _sleepon_lock(),
     _sleepon_signal(), _sleepon_unlock()
Destroy a sleepon

Synopsis:

```c
#include <pthread.h>

int _sleepon_destroy( sleepon_t * l );
```

Arguments:

- `l`: A pointer to a `sleepon_t` that you created by calling `_sleepon_init()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_destroy()` function destroys a `sleepon_t` structure, `l`, that has been previously initialized by `_sleepon_init()`.

If `l` hasn’t been locked by `_sleepon_lock()`, `_sleepon_destroy()` locks it before destroying it.

The sleepon structure is reference-counted such that, if other threads are blocked waiting for a condition, they’re be signaled to wake up, and the last one to wake up frees the memory allocated to the sleepon.

Returns:

- `0`: Success.
- `≠0`: Failure

Classification:

QNX Neutrino
See also:

_sleepon_broadcast(), _sleepon_init(), _sleepon_lock(),
_sleepon_signal(), _sleepon_unlock()
**_sleepon_init_()**

*Initialize a sleepon*

**Synopsis:**

```c
#include <pthread.h>

int _sleepon_init( sleepon_t **pl,
                      unsigned flags);
```

**Arguments:**

- `pl` The address of a location where the function can store a pointer to the `sleepon_t` object that it creates.
- `flags` There are currently no flags defined; pass zero for this argument.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `_sleepon_init()` function allocates a `sleepon_t` object (which is an opaque data structure) and stores a pointer to it in the location that `pl` points to.

**Returns:**

- `0` Success.
- `≠0` Failure.

**Classification:**

QNX Neutrino
### _sleepon_init()  

#### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

#### See also:

- _sleepon_broadcast()
- _sleepon_destroy()
- _sleepon_lock()
- _sleepon_signal()
- _sleepon_unlock()
**_sleepon_lock_**

Lock a sleepon

**Synopsis:**
```
#include <pthread.h>

int _sleepon_lock( sleepon_t * l );
```

**Arguments:**

- `l`: A pointer to a `sleepon_t` that you created by calling `_sleepon_init()`.

**Library:**

-libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `_sleepon_lock()` function locks the mutex associated with the sleepon structure, `l`.

You must call this function before calling `_sleepon_wait()`, `_sleepon_signal()`, or `_sleepon_broadcast()`.

**Returns:**

- **EOK**: Success.
- **EAGAIN**: Insufficient system resources were available to lock the mutex.
- **EDEADLK**: The calling thread already owns `mutex`, and the mutex doesn’t allow recursive behavior.
- **EINVAL**: Invalid mutex.

The `_sleepon_lock()` function returns the same values as `pthread_mutex_lock()`.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_lock(), _sleepon_broadcast(), _sleepon_destroy(), _sleepon_init(), _sleepon_signal(), _sleepon_unlock(), _sleepon_wait()
**_sleepon_signal()**  
Wake up a single thread

**Synopsis:**
```c
#include <pthread.h>

int _sleepon_signal( sleepon_t * l,
                     const volatile void * addr );
```

**Arguments:**
- `l` is a pointer to a `sleepon_t` that you created by calling `_sleepon_init()`.
- `addr` is the handle that the thread is waiting on. The value of `addr` is typically a data structure that controls a resource.

**Library:**
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `_sleepon_signal()` and `_sleepon_broadcast()` functions are very similar:

- `_sleepon_signal()` wakes up a single thread that’s waiting on the key, `addr`.
- `_sleepon_broadcast()` wakes up all threads that are waiting on the key, `addr`.

The waiting threads must have used the same sleepon, `l` and key, `addr`, in order to be woken up.

To be woken up, the calling threads must have been locked by `_sleepon_lock()`.
Returns:

0  Success.
≠0  Failure.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_sleepon_broadcast(), pthread_sleepon_signal(), _sleepon_broadcast(), _sleepon_destroy(), _sleepon_init(), _sleepon_lock(), _sleepon_unlock()
Unlock a sleepon

Synopsis:
```
#include <pthread.h>

int _sleepon_unlock( sleepon_t * l );
```

Arguments:

l A pointer to a `sleepon_t` that you created by calling `_sleepon_init()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `_sleepon_unlock()` function unlocks the mutex associated with the sleepon structure, `l`. You must have previously locked the mutex by calling `_sleepon_lock()`.

Returns:

- EOK Success.
- EINVAL Invalid mutex `mutex`.
- EPERM The current thread doesn’t own `mutex`.

The `_sleepon_unlock()` function returns the same values as `pthread_mutex_unlock()`.

Classification:

QNX Neutrino
_sleepon_unlock()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_unlock(), _sleepon_broadcast(), _sleepon_destroy(), _sleepon_init(), _sleepon_lock()
Wait on a sleepon lock

Synopsis:
#include <pthread.h>

int _sleepon_wait( sleepon_t * l,
        const volatile void * addr,
        _uint64 nsec);

Arguments:
l A pointer to a sleepon_t that you created by calling
    _sleepon_init().
addr The handle that you want to wait on. The value of addr is
typically a data structure that controls a resource.
nsec Zero, or the amount of time, in nanoseconds, to wait before
the thread wakes up. If this timeout occurs, ETIMEDOUT is
returned.

Library:
libc
Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:
The _sleepon_wait() function blocks on the sleepon l using the key
addr until woken up by either a _sleepon_signal() or a
_sleepon_broadcast() call that uses the same addr key.

The calling thread must first have locked the sleepon by calling
_sleepon_lock().

When the thread returns from this function, it must release the
sleepon lock by calling _sleepon_unlock().
Returns:

0    Success.
≠0   Failure; a nonzero `errno` value.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`_sleepon_broadcast()`, `_sleepon_lock()`, `_sleepon_unlock()`, `_sleepon_signal()`
slogb()

Send a message to the system logger

Synopsis:

```c
#include <stdio.h>
#include <sys/slog.h>

int slogb( int code,
           int severity,
           void * data,
           int size );
```

Arguments:

- **opcode**: A combination of a *major* and *minor* code.
- **severity**: The severity of the log message; see “Severity levels,” in the documentation for `slogf()`.
- **data**: A block of raw data.
- **size**: The size of the raw data.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

---

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `slog*()` functions send log messages to the system logger, `slogger`. To send formatted messages, use `slogf()`. If you have programs that scan log files for specified codes, you can use `slogb()` or `slogi()` to send a block of structures or `int`'s, respectively.
Errors:

Any value from the Errors section in `MsgSend()`, as well as:

- **EACCESS** Insufficient permission to write to the log file.
- **EINVAL** The size of the data buffer exceeds 255×4 bytes, or an odd number of bytes is being read.
- **ENOENT** Invalid log file or directory specified, or `slogger` isn’t running.

Examples:

See `slogf()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`slogf()`, `slogi()`, `vslogf()`

`slogger`, `sloginfo` in the *Utilities Reference*
Send a message to the system logger

Synopsis:

```
#include <stdio.h>
#include <sys/slog.h>

int slogf( int opcode,  
           int severity,  
           const char *fmt, 
           ...);
```

Arguments:

- `opcode` - A combination of a major and minor code. Create the `opcode` using the `_SLOG_SETCODE(major, minor)` macro that's defined in `<sys/slog.h>`. The `major` and `minor` codes are defined in `<sys/slogcodes.h>`.

- `severity` - The severity of the log message; see “Severity levels,” below.

- `fmt` - A standard `printf()` string followed by `printf()` arguments.

The formatting characters that you use in the message determine any additional arguments.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).
**slogf()**

**Description:**

The `slog*()` functions send log messages to the system logger, `slogger`. To send formatted messages, use `slogf()`. If you have programs that scan log files for specified codes, you can use `slogb()` or `slogi()` to send a block of structures or `int`’s, respectively.

The `vslogf()` function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of `<stdarg.h>`.

### Severity levels

There are eight levels of severity defined. The lowest severity is 7 and the highest is 0. The default is 7.

<table>
<thead>
<tr>
<th>Manifest Name</th>
<th>Severity value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SLOG_SHUTDOWN</td>
<td>0</td>
<td>Shut down the system NOW (e.g. for OEM use)</td>
</tr>
<tr>
<td>_SLOG_CRITICAL</td>
<td>1</td>
<td>Unexpected unrecoverable error (e.g. hard disk error)</td>
</tr>
<tr>
<td>_SLOG_ERROR</td>
<td>2</td>
<td>Unexpected recoverable error (e.g. needed to reset a hardware controller)</td>
</tr>
<tr>
<td>_SLOG_WARNING</td>
<td>3</td>
<td>Expected error (e.g. parity error on a serial port)</td>
</tr>
<tr>
<td>_SLOG_NOTICE</td>
<td>4</td>
<td>Warnings (e.g. out of paper)</td>
</tr>
<tr>
<td>_SLOG_INFO</td>
<td>5</td>
<td>Information (e.g. printing page 3)</td>
</tr>
</tbody>
</table>

*continued...*
slogf()

<table>
<thead>
<tr>
<th>Manifest Name</th>
<th>Severity value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SLOG_DEBUG1</td>
<td>6</td>
<td>Debug messages (normal detail)</td>
</tr>
<tr>
<td>_SLOG_DEBUG2</td>
<td>7</td>
<td>Debug messages (fine detail)</td>
</tr>
</tbody>
</table>

**Returns:**

The size of the message sent to `slogger`, or -1 if an error occurs.

**Errors:**

Any value from the Errors section in `MsgSend()`, as well as:

- **EACCES** Insufficient permission to write to the log file.
- **EINVAL** The size of the data buffer exceeds 255×4 bytes, or an odd number of bytes is being read.
- **ENOENT** Invalid log file or directory specified, or `slogger` isn’t running.

**Examples:**

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/slog.h>
#include <sys/slogcodes.h>

int main() {
    int i;

    for(i = 0 ; ; i++) {
        switch(rand() % 3) {
            case 0:
                slogb(_SLOG_SETCODE(_SLOGC_TEST, 0),
                      _SLOG_DEBUG1, &i, sizeof(i));
                break;

            case 1:
                slogi(_SLOG_SETCODE(_SLOGC_TEST, 1),
                      _SLOG_CRITICAL, 1, i);
                break;
        }
    }
}
```

3064 C Library — S September 10, 2007
```c
    case 2:
        slogf( _SLOG_SETCODE(_SLOGC_TEST, 2),
               _SLOG_ERROR,
               "This is number \d", i);
        break;
    }
    sleep(1);
}
```

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`slogb()`, `slogi()`, `vslogf()`

`slogger`, `sloginfo` in the *Utilities Reference*
slogi()

Send a message to the system logger

Synopsis:

```c
#include <stdio.h>
#include <sys/slog.h>

int slogi( int code,  
           int severity,  
           int nargs,  
           ... );
```

Arguments:

- **opcode**: A combination of a *major* and *minor* code.
- **severity**: The severity of the log message; see “Severity levels,” in the documentation for `slogf()`.
- **nargs**: The number of integers to send. A maximum of 32 integers is allowed.

The additional arguments are the integers that you want to write.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

Description:

The `slog*()` functions send log messages to the system logger, `slogger`. To send formatted messages, use `slogf()`. If you have programs that scan log files for specified codes, you can use `slogb()` or `slogi()` to send a block of structures or `int`'s, respectively.
Errors:

Any value from the Errors section in `MsgSend()`, as well as:

- **EACCES** Insufficient permission to write to the log file.
- **EINVAL** The size of the data buffer exceeded 32 integers.
- **ENOENT** Invalid log file or directory specified, or `slogger` isn’t running.

Examples:

See `slogf()`

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`slogb()`, `slogf()`, `slogi()`, `vsllogf()`

`slogger`, `sloginfo` in the `Utilities Reference`
**_smalloc()**

Allocate memory in blocks

**Synopsis:**
```
#include <malloc.h>

void* _smalloc( size_t size );
```

**Arguments:**

- `size` The size of the block to allocate, in bytes.

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The _smalloc() function allocates space for an object of `size` bytes. Nothing is allocated when the `size` argument has a value of zero.

This function allocates memory in blocks of _amblksiz bytes; _amblksiz is a global variable defined in `<stdlib.h>`.

You must use _sfree() to deallocate the memory allocated by _smalloc().

**Returns:**

A pointer to the start of the allocated memory, or NULL if there’s insufficient memory available, or if the requested `size` is zero.
_smallloc()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

calloc(), free(), realloc(), _scalloc(), _sfree(), _srealloc()
**Synopsis:**

```c
#include <sys/types.h>
#include <snmp/asn1.h>
#include <snmp/snmp_api.h>

extern int snmp_errno

int snmp_close( struct snmp_session * session );
```

**Arguments:**

- `session`: A pointer to the `snmp_session` structure that identifies the SNMP session that you want to close. This pointer was returned by a call to `snmp_open()`.

**Library:**

`libsnp`

Use the `-l snmp` option to `qcc` to link against this library.

**Description:**

The `snmp_close()` function closes the input session, frees any data allocated for it, dequeues any pending requests, and closes any sockets allocated for the session.

**Returns:**

- `1`: Success.
- `0`: An error occurred (`snmp_errno` is set).

**Errors:**

If an error occurs, this function sets `snmp_errno` to:

- `SNMPERR_BAD_SESSION`

  The specified session wasn’t open.
snmp_close()

Classification:

SNMP

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

select(), snmp_free_pdu(), snmp_open(), snmp_pdu,
snmp_pdu_create(), snmp_read(), snmp_select_info(), snmp_send(),
snmp_session, snmp_timeout()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

Free an SNMP Protocol Data Unit message structure

**Synopsis:**

```c
#include <sys/types.h>
#include <netinet/in.h>
#include <snmp/asn1.h>
#include <snmp/snmp_api.h>

void snmp_free_pdu( struct snmp_pdu * pdu );
```

**Arguments:**

- `pdu` A pointer to the `snmp_pdu` structure that you want to free.

**Library:**

`libsnmp`

Use the `-l snmp` option to `qcc` to link against this library.

**Description:**

The `snmp_free_pdu()` function frees the `snmp_pdu` structure pointed to by `pdu`, and any data that it contains that was allocated with `malloc()`.

**Classification:**

SNMP

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: No
See also:

`select()`, `snmp_close()`, `snmp_open()`, `snmp_pdu`, `snmp_read()`, `snmp_select_info()`, `snmp_send()`, `snmp_session`, `snmp_timeout()`

RFC 1157, FAQ in Internet newsgroup `comp.protocols.snmp`

**snmp_open()**

Open an SNMP session

**Synopsis:**

```c
#include <sys/types.h>
#include <snmp/asn1.h>
#include <snmp/snmp_api.h>

extern int snmp_errno;

struct snmp_session * snmp_open(
    struct snmp_session * session );
```

**Arguments:**

*session* A pointer to a `snmp_session` structure that defines the SNMP session that you want to open.

**Library:**

`libsnmp`

Use the `-l snmp` option to `qcc` to link against this library.

**Description:**

The `snmp_open()` function sets up an SNMP session with the information supplied by the application in the `snmp_session` structure pointed to by `session`. Next, `snmp_open()` opens and binds the necessary UDP port.

**Returns:**

A pointer to a `snmp_session` structure for the created session (which is different from the pointer passed to the function), or NULL if an error occurs (`snmp_errno` is set).

**Errors:**

If an error occurs, this function sets `snmp_errno` to one of:

- `SNMPERR_BAD_ADDRESS`
  - Unknown host.
SNMPERR_BAD_LOCPORT
   Couldn’t bind to the specified port.
SNMPERR_GENERR
   Couldn’t open the socket.

Classification:
SNMP

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

read_main_config_file(), select(), snmp_close(), snmp_free_pdu(), snmp_pdu, snmp_pdu_create(), snmp_read(), snmp_select_info(), snmp_send(), snmp_session, snmp_timeout()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

**Synopsis:**

```
#include <snmp/snmp_api.h>

struct snmp_pdu {
    int version;
    ipaddr address;
    oid * srcParty;
    int srcPartyLen;
    oid * dstParty;
    int dstPartyLen;
    oid * context;
    int contextLen;
    u_char * community;
    int community_len;
    int command;
    long reqid;
    long errstat;
    long errindex;

    /* Trap information */
    oid * enterprise;
    int enterprise_length;
    ipaddr agent_addr;
    int trap_type;
    int specific_type;
    u_long time;

    struct variable_list * variables;
};
```

**Description:**

The `snmp_pdu` structure describes a Protocol Data Unit (PDU), a transaction that’s performed over an open session. It contains the headers and variables of an SNMP packet. The structure includes the following members:

- **version**: The version of SNMP: either `SNMP_VERSION_1` or `SNMP_VERSION_2`.
- **address**: The destination IP address.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcParty</td>
<td>The source party being used.</td>
</tr>
<tr>
<td>srcPartyLen</td>
<td>The number of object identifier (OID) elements in srcParty. For example, if srcParty is <code>.1.3.6</code>, the length is 3.</td>
</tr>
<tr>
<td>dstParty</td>
<td>The destination party being used.</td>
</tr>
<tr>
<td>dstPartyLen</td>
<td>The number of OID elements in dstParty.</td>
</tr>
<tr>
<td>context</td>
<td>The context being used.</td>
</tr>
<tr>
<td>contextLen</td>
<td>The number of OID elements in context.</td>
</tr>
<tr>
<td>community</td>
<td>The community for outgoing requests.</td>
</tr>
<tr>
<td>community_len</td>
<td>The length of the community name.</td>
</tr>
<tr>
<td>command</td>
<td>The type of this PDU.</td>
</tr>
<tr>
<td>reqid</td>
<td>The request ID. The default is SNMP_DEFAULT_REQID (0).</td>
</tr>
<tr>
<td>errstat</td>
<td>The error status (non_repeaters in GetBulk). The default is SNMP_DEFAULT_ERRSTAT (-1).</td>
</tr>
<tr>
<td>errindex</td>
<td>The error index (max_repetitions in GetBulk). The default is SNMP_DEFAULT_ERRINDEX (-1).</td>
</tr>
<tr>
<td>enterprise</td>
<td>The system OID.</td>
</tr>
<tr>
<td>enterprise_len</td>
<td>The number of OID elements in enterprise. The default is SNMP_DEFAULT_ENTERPRISE_LENGTH (0).</td>
</tr>
<tr>
<td>agent_addr</td>
<td>The address of the object generating the trap.</td>
</tr>
<tr>
<td>trap_type</td>
<td>The trap type.</td>
</tr>
<tr>
<td>specific_type</td>
<td>The specific type.</td>
</tr>
</tbody>
</table>
Time

The up time. The default is SNMP_DEFAULT_TIME (0).

Variables

A linked list of variables, of type variable_list.

The variable_list structure is defined as:

```c
typedef struct sockaddr_in ipaddr;

struct variable_list {
    struct variable_list* next_variable;
    oid* name;
    int name_length;
    u_char type;
    union {
        long* integer;
        u_char* string;
        oid* objid;
        u_char* bitstring;
        struct counter64* counter64;
    } val;
    int val_len;
};
```

The members are:

- **next_variable**
  A pointer to the next variable. This is NULL for the last variable in the list.

- **name**
  The object identifier of the variable.

- **name_length**
  The number of sub IDs in name.

- **type**
  The ASN type of variable.

- **val.integer**
  The value of the variable if it's an integer.

- **val.string**
  The value of the variable if it's a string.

- **val.objid**
  The value of the variable if it's an object ID.

- **bitstring**
  The value of the variable if it's a bitstring.

- **counter64**
  The value of the variable if it's a counter64.

- **val_len**
  The length of the value.
Classification:

SNMP

See also:

snmp_close(), snmp_free_pdu(), snmp_open(), snmp_pdu_create(), snmp_read(), snmp_send(), snmp_session

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

**snmp_pdu_create()**

Create an SNMP Protocol Data Unit message structure

**Synopsis:**

```
#include <sys/types.h>
#include <netinet/in.h>
#include <snmp/snmp.h>
#include <snmp/asn1.h>
#include <snmp/snmp_api.h>
#include <snmp/snmp_client.h>

extern int snmp_errno;

struct snmp_pdu * snmp_pdu_create (int command);
```

**Arguments:**

- `command` The type of message that the PDU represents:
  - BULK_REQ_MSG
  - GET_REQ_MSG
  - GET_RSP_MSG
  - GETNEXT_REQ_MSG
  - INFORM_REQ_MSG
  - SET_REQ_MSG
  - TRP_REQ_MSG
  - TRP2_REQ_MSG
  
  as defined in `<snmp/snmp.h>`.

**Library:**

```
libsnpmp
```

Use the `-l snmp` option to `qcc` to link against this library.

**Description:**

The `snmp_pdu_create()` function allocates memory for a Protocol Data Unit (PDU) structure for SNMP message passing. The PDU structure is initialized with default values; see the documentation for `snmp_pdu`. 
Returns:

A pointer to the PDU structure created, or NULL if an error occurs
\(\text{snmp_errno}\) is set).

Errors:

If an error occurs, this function sets \text{snmp_errno} to:

\text{SNMPERR\_GENERR}

Not enough memory to create the \text{snmp\_pdu} structure.

Classification:

SNMP

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

\text{snmp\_free\_pdu()}, \text{snmp\_pdu}, \text{snmp\_read()}, \text{snmp\_send()}

\text{RFC 1157}, FAQ in Internet newsgroup \text{comp.protocols.snmp}

**snmp_read()**

*Read an SNMP message*

**Synopsis:**

```c
#include <sys/select.h>
#include <snmp/snmp_impl.h>

void snmp_read( struct fd_set * fdset );
```

**Arguments:**

- `fdset` A pointer to a `fd_set` structure that contains all the file descriptors that you want to read from.

**Library:**

- `libsnmp`

  Use the `-l snmp` option to `qcc` to link against this library.

**Description:**

The `snmp_read()` function reads a packet from each socket and its set of file descriptors and parses the packet. The resulting Protocol Data Unit (PDU) is passed to the callback routine for the session (see `snmp_session`); if the callback returns successfully, the PDU and its request are deleted.

For information on asynchronous SNMP transactions, see `snmp_select_info()`.

**Classification:**

- SNMP

  **Safety**
  
<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

  *continued...*
See also:

select(), snmp_close(), snmp_open(), snmp_pdu, snmp_read(),
snmp_select_info(), snmp_send(), snmp_session, snmp_timeout()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

Marshall T. Rose, The Simple Book: An Introduction to Internet
0-13-451659-1)
**Synopsis:**

```c
#include <sys/types.h>
#include <sys/select.h>
#include <sys/time.h>
#include <snmp/snmp_api.h>

int snmp_select_info( int * numfds,
                      struct fd_set * fdset,
                      struct timeval * timeout,
                      int * block );
```

**Arguments:**

- `numfds` The number of significant file descriptors in `fdset`.
- `fdset` A pointer to a set of file descriptors that contains all of the file descriptors that you’ve opened for SNMP. If activity occurs on any of these file descriptors, you should call `snmp_read()` with that file-descriptor set.
- `timeout` A pointer to a `timeval` structure that defines the longest time that SNMP can wait for a timeout. You should call `select()` with the minimum time between `timeout` and any other timeouts necessary. You should check this on each invocation of `select()`. If a timeout is received, you should call `snmp_timeout()` to see if the timeout was for SNMP. (The `snmp_timeout()` function is idempotent.)
  
  You must provide the `timeout`, even if `block` is 1 (see below).
- `block` Governs the behavior of `select()`:
  
  - If `block` is 0, `select()` is requested to time out.
  - If `block` is 1, `select()` is requested to block indefinitely. The timeout value is treated as undefined, although you must provide it. On return, if `block` is nonzero, the value of `timeout` is undefined.
**Library:**

\[\text{libsnmp}\]

Use the \texttt{-l snmp} option to \texttt{qcc} to link against this library.

**Description:**

The \texttt{snmp\_select\_info()} function is used to return information about what SNMP requires from a \texttt{select()} call.

**Asynchronous SNMP transactions:**

To have SNMP transactions occur asynchronously, you can invoke the functions \texttt{snmp\_select\_info()}, \texttt{snmp\_timeout()}, and \texttt{snmp\_read()} in conjunction with the system call \texttt{select()}. For more information, see \texttt{select()}. For asynchronous transactions, invoke \texttt{snmp\_select\_info()} with the information you would have passed to \texttt{select()} in the absence of SNMP. The \texttt{snmp\_select\_info()} function modifies the information, which is subsequently passed to \texttt{select()}.

**Parameters to \texttt{select()}:**

| \texttt{nfds} | \texttt{numfds} |
| \texttt{readfds} | \texttt{fdset} |
| \texttt{timeout} | \texttt{timeout}—must point to an allocated (but not necessarily initialized) \texttt{timeval} structure. |

The following code segment shows how to use these SNMP functions in conjunction with \texttt{select()}:  

```c
FD_ZERO(&fdset);
numfds=sd+1;
FD_SET(sd,&fdset);
block=0;
tvp=&timeout;
timerclear(tvp);
```
snmp_select_info()

```c

tvp->tv_sec = 5;

snmp_select_info(&numfds, &fdset, tvp, &block);

if(block==1)
{
  tvp = NULL;
}

count = select(numfds, &fdset, 0, 0, tvp);

if(count==0)
  snmp_timeout();
if(count>0)
  snmp_read(&fdset);
```

**Returns:**

The number of open sockets (i.e. the number of open sessions).

**Classification:**

SNMP

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`select()`, `snmp_close()`, `snmp_open()`, `snmp_pdu`, `snmp_read()`, `snmp_select_info()`, `snmp_send()`, `snmp_session`, `snmp_timeout()`

RFC 1157, FAQ in Internet newsgroup `comp.protocols.snmp`

Synopsis:

```c
#include <sys/types.h>
#include <snmp/asn1.h>
#include <netinet/in.h>
#include <snmp/snmp_api.h>

extern int snmp_errno;

int snmp_send( struct snmp_session * session, struct snmp_pdu * pdu );
```

Arguments:

- `session` A pointer to the `snmp_session` structure that identifies the SNMP session that you want to send the message on. This pointer was returned by a call to `snmp_open()`.
- `pdu` A pointer to the `snmp_pdu` structure that defines the Protocol Data Unit that you want to send. Create this structure by calling `snmp_pdu_create()`.

Library:

```
libsnmp
```

Use the `-l snmp` option to `qcc` to link against this library.

Description:

The `snmp_send()` function sends the PDU on the session provided. If necessary, some of the `snmp_pdu` structure data is set from the session defaults. A request corresponding to this PDU is added to the list of outstanding requests on this session and then the packet is sent.

This function frees `pdu` unless an error occurs.

Returns:

The request ID of the generated packet, if applicable, 1 if not applicable, or 0 if an error occurs (`snmp_errno` is set).
Errors:

If an error occurs, this function sets `snmp_errno` to one of:

SNMPERR_BAD_ADDRESS

A necessary entity in the `pdu` structure was omitted. These include:

- `version`
- `address` and the `snmp_session peername` member
- `srcParty` (SNMP version 2 only)
- `dstParty` (SNMP version 2 only)
- `context` (SNMP version 2 only)
- `community_len` (SNMP version 1 only)

SNMPERR_BAD_SESSION

The specified session wasn’t open.

SNMPERR_GENERR

An error occurred forming the packet.

Classification:

SNMP

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>
See also:

select(), snmp_close(), snmp_open(), snmp_pdu,
snmp_pdu_create(), snmp_read(), snmp_select_info(), snmp_send(),
snmp_session, snmp_timeout()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

Marshall T. Rose, The Simple Book: An Introduction to Internet
0-13-451659-1)
Synopsis:
#include <snmp/snmp_api.h>

struct snmp_session {
    u_char * community;
    int community_len;
    int retries;
    long timeout;
    char * peername;
    u_short remote_port;
    u_short local_port;
    u_char * (*authenticator)();
    int (*callback)();
    void * callback_magic;
    int version;
    oid * srcParty;
    int srcPartyLen;
    oid * dstParty;
    int dstPartyLen;
    oid * context;
    int contextLen;
};

Description:

The snmp_session structure describes a set of transactions sharing similar transport characteristics. It includes the following members:

*community* The community for outgoing requests. The default is 0.

*community_len* The length of the community name. The default is SNMP_DEFAULT_COMMUNITY_LEN (0).

*retries* The number of retries before timing out. The default is SNMP_DEFAULT_RETRIES (-1).

*timeout* The number of microseconds until the first timeout. Subsequent timeouts increase exponentially. The default is SNMP_DEFAULT_TIMEOUT (-1).
peername

The domain name or dotted IP address of the default peer. The default is SNMP_DEFAULT_PEERNAME (NULL).

remote_port

The UDP port number of the peer. The default is SNMP_DEFAULT_REMPORT (0).

local_port

My UDP port number. The default is SNMP_DEFAULT_ADDRESS (0), for picked randomly.

authenticator

The authentication function, or NULL if null authentication is used. If your application is using version 1 of SNMP, you must supply this member.

The authenticator() function is defined as:

```
 u_char* authenticator( u_char* pdu,
                        int* length,
                        u_char* community,
                        int community_len)
```

The arguments are:

- `pdu` — the rest of the PDU to be authenticated.
- `length` — the length of the remaining data in the PDU, updated by authenticator().
- `community` — the community name for authentication.
- `community_len` — the length of the community name.

To specify null authentication, set the authenticator field in `snmp_session` to NULL.

The authenticator() function returns an authenticated PDU, or NULL if an error occurs.

callback

A function used to extract the data from the received packet (the `snmp_pdu` structure passed to the callback). The application must supply this member.
The `callback()` function is defined as:

```c
int callback( int operation,
             struct snmp_session* session,
             int reqid,
             struct snmp_pdu* pdu,
             void* magic);
```

The arguments are:

- `operation` — the possible operations are `RECEIVED_MESSAGE` and `TIMED_OUT`.
- `session` — the session that was authenticated using `community`.
- `reqid` — the request ID identifying the transaction within this session. Use 0 for traps.
- `pdu` — A pointer to PDU information. You must copy the information because it will be freed elsewhere.
- `magic` — a pointer to the data for `callback()`.

The callback should return 1 on successful completion, or 0 if it should be kept pending.

`callback_magic` — A pointer to data that the callback function may consider important.

`version` — The version of SNMP: either `SNMP_VERSION_1` or `SNMP_VERSION_2`.

`srcParty` — The source party being used for this session.

`srcPartyLen` — The number of object identifier (OID) elements in `srcParty`. For example, if `srcParty` is `.1.3.6`, the length is 3.

`dstParty` — The destination party being used for this session.

`dstPartyLen` — The number of OID elements in `dstParty`.

`context` — The context being used for this session.

`contextLen` — The number of OID elements in `context`. 
Classification:

SNMP

See also:

snmp_close(), snmp_free_pdu(), snmp_open(), snmp_pdu, snmp_send()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

Timeout during an SNMP session

Synopsis:

```c
#include <snmp/snmp_api.h>

void snmp_timeout( void );
```

Library:

```
libsnmp
```

Use the `-l snmp` option to `qcc` to link against this library.

Description:

The `snmp_timeout()` function handles any outstanding SNMP requests. It should be called whenever the timeout from `snmp_select_info()` expires. The `snmp_timeout()` function checks to see if any of the sessions has an outstanding request that has timed out.

If it finds one or more, and that PDU has more retries available, a new packet is formed from the PDU and is resent. If there are no more retries available, the callback for the session is used to alert the user of the timeout by setting the callback’s `operation` argument to `TIMED_OUT` (2).

For information on asynchronous SNMP transactions, see `snmp_select_info()`.

Classification:

```
SNMP
```

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
</tbody>
</table>

continued...
snmp_timeout()

Safety
Thread No

See also:
select(), snmp_close(), snmp_open(), snmp_pdu, snmp_read(), snmp_select_info(), snmp_send(), snmp_session, snmp_timeout()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

**Synopsis:**

```c
#include <stdio.h>

int snprintf( char* buf,
              size_t count,
              const char* format,
              ...);
```

**Arguments:**

- `buf` A pointer to the buffer where you want the function to store the formatted string.
- `count` The maximum number of characters to store in the buffer, including a terminating null character.
- `format` A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `snprintf()` function is similar to `fprintf()`, except that `snprintf()` places the generated output into the character array pointed to by `buf`, instead of writing it to a file. A null character is placed at the end of the generated character string.

**Returns:**

The number of characters that would have been written into the array, not counting the terminating null character, had `count` been large enough. It does this even if `count` is zero; in this case `buf` can be NULL.

If an error occurred, `snprintf()` returns a negative value and sets `errno`. 
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

/* Create temporary file names using a counter */

char namebuf[13];
int TempCount = 0;

char *make_temp_name( void )
{
    snprintf( namebuf, 13, "ZZ%.6o.TMP",
                TempCount++ );
    return( namebuf );
}

int main( void )
{
    FILE *tf1, *tf2;

    tf1 = fopen( make_temp_name(), "w" );
    tf2 = fopen( make_temp_name(), "w" );
    fputs( "temp file 1", tf1 );
    fputs( "temp file 2", tf2 );
    fclose( tf1 );
    fclose( tf2 );

    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**Caveats:**

Be careful if you’re using `snprintf()` to build a string one piece at a time. For example, this code:

```c
len += snprintf(&buf[len], RECSIZE - 1 - len, ...);
```

could have a problem if `snprintf()` truncates the string. Without a separate test to compare `len` with `RECSIZE`, this code doesn’t protect against a buffer overflow. After the call that truncates the output, `len` is larger than `RECSIZE`, and `RECSIZE - 1 - len` is a very large (unsigned) number; the next call generates unlimited output somewhere beyond the buffer.

**See also:**

`errno`, `fprintf()`, `fwprintf()`, `printf()`, `sprintf()`, `swprintf()`, `vfprintf()`, `vfscanf()`, `vprintf()`, `vsnprintf()`, `vsprintf()`, `vswprintf()`, `vwprintf()`, `wprintf()`
sockatmark()

Determine whether a socket is at the out-of-band mark

Synopsis:

```c
#include <sys/socket.h>

int sockatmark( int s );
```

Arguments:

- `s`  
The file descriptor of the socket that you want to check, as returned by `socket()`.

Library:

- **libsocket**
  
  Use the `-l socket` option to `qcc` to link against this library.

Description:

The `sockatmark()` function determines whether the socket specified by `s` is at the out-of-band data mark. If the protocol for the socket supports out-of-band data by marking the stream with an out-of-band data mark, `sockatmark()` returns 1 when all data preceding the mark has been read and the out-of-band data mark is the first element in the receive queue.

The `sockatmark()` function doesn’t remove the out-of-band data mark from the stream.

Using this function between receive operations lets an application determine which data comes before and after out-of-band data.

Returns:

- 0 The socket isn’t at the out-of-band data mark.
- 1 The socket is at the out-of-band data mark.
- -1 An error occurred (`errno` is set).
Errors:

EBADF    Invalid file descriptor s.
ENOTTY   The s argument isn’t the file descriptor of a valid socket.

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

recv(), recvmsg()
socket()

Create an endpoint for communication

Synopsis:

```c
#include <sys/types.h>
#include <sys/socket.h>

int socket( int domain,
            int type,
            int protocol );
```

Arguments:

domain The communications domain that you want to use. This selects the protocol family that should be used. These families are defined in `<sys/socket.h>`.

type The type of socket you want to create. This determines the semantics of communication. Here are the currently defined types:

- SOCK_STREAM — provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.
- SOCK_DGRAM — supports datagrams, which are connectionless, unreliable messages of a fixed (typically small) maximum length.
- SOCK_RAW — provides access to internal network protocols and interfaces. Available only to the superuser, this type isn’t described here.

For more information, see below.

protocol The particular protocol that you want to use with the socket. Normally, only a single protocol exists to support a particular socket type within a given protocol family. But if many protocols exist, you must specify one. The protocol number you give is particular to the communication domain where communication is to take place (see `/etc/protocols` in the Utilities Reference).
Library:

libsocket

Use the -l socket option to gcc to link against this library.

Description:

The socket() function creates an endpoint for communication and returns a descriptor.

SOCK_STREAM sockets

SOCK_STREAM sockets are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. To create a connection to another socket, call connect() call.

Once the socket is connected, you can transfer data by using read() and write() or some variant of send() and recv(). When a session has been completed, a close() may be performed. Out-of-band data may also be transmitted (as described in send()) and received (as described in recv()).

The communications protocols used to implement a SOCK_STREAM socket ensure that data isn’t lost or duplicated. If a piece of data that the peer protocol has buffer space for can’t be successfully transmitted within a reasonable length of time, the connection is considered broken and calls will indicate an error by returning -1 and setting errno to ETIMEDOUT.

SOCK_DGRAM and SOCK_RAW sockets

With SOCK_DGRAM and SOCK_RAW sockets, datagrams can be sent to correspondents named in send() calls. Datagrams are generally received with recvfrom(), which returns the next datagram with its return address.
Using the `ioctl()` call

You can use the `ioctl()` call to specify a process group to receive a SIGURG signal when the out-of-band data arrives. The call may also enable nonblocking I/O and asynchronous notification of I/O events via SIGIO.

Socket-level options

The operation of sockets is controlled by socket-level options. These options are defined in the file `<sys/socket.h>`. Use `setsockopt()` and `getsockopt()` to set and get options.

Returns:

A descriptor referencing the socket, or -1 if an error occurs (`errno` is set).

Errors:

- **EACCES**  
  Permission to create a socket of the specified type and/or protocol is denied.
- **EMFILE**  
  The per-process descriptor table is full.
- **ENFILE**  
  The system file table is full.
- **ENOBUFFS**  
  Insufficient buffer space available. The socket can’t be created until sufficient resources are freed.
- **ENOMEM**  
  Not enough memory.
- **EPROTONOSUPPORT**  
  The protocol type or the specified protocol isn’t supported within this domain.

Classification:

POSIX 1003.1
By default, `socket()` communicates with the TCP/IP stack managing the `/dev/socket` portion of the namespace. You can change this behavior by setting the `SOCK` environment variable. For an example, see “Running multiple instances of the TCP/IP stack” in the TCP/IP Networking chapter of the Neutrino User’s Guide.

See also:

ICMP6, ICMP, INET6, IPv6, IP, IPsec, ROUTE, TCP, UDP, UNIX protocols

`accept()`, `bind()`, `close()`, `connect()`, `getprotobynumber()`, `getsockname()`, `getsockopt()`, `ioctl()`, `listen()`, `read()`, `recv()`, `select()`, `send()`, `shutdown()`, `socketpair()`, `write()`
socketpair()

Create a pair of connected sockets

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair( int domain,
                int type,
                int protocol,
                int * fd[2] );
```

Arguments:

- `domain` The communications domain where the sockets are to be created.
- `type` The type of sockets to create.
- `protocol` The protocol to use with the sockets. A protocol of 0 causes `socketpair()` to use an unspecified default protocol appropriate for the requested socket type.
- `fd[2]` The 2-digit integer array where the file descriptors of the created socket pair are to be held.

Library:

`libsocket`

Use the `-l socket` option to `qcc` to link against this library.

Description:

The `socketpair()` call creates an unnamed pair of connected sockets in the specified `domain`, of the specified `type`, using the optionally specified `protocol` argument. The file descriptors are returned in the vector `fd` and are identical.

Valid types are described in `socket()`.

If the `protocol` argument is nonzero, it must be a protocol that’s understood by the address family. No such protocols are defined at this time.
socketpair()

Returns:

0  Success.

-1  An error occurred (errno is set).

Errors:

EAFNOSUPPORT
The specified address family isn’t supported on this machine.

EFAULT
The address sv doesn’t specify a valid part of the process address space.

EMFILE
Too many descriptors are in use by this process.

EOPNOTSUPP
The specified protocol doesn’t support creation of socket pairs.

EPROTONOSUPPORT
The specified protocol isn’t supported on this machine.

Examples:

```c
#include <stdio.h>
#include <sys/socket.h>

#define CHAR_BUFSIZE 20
int main(int argc, char **argv) {
    int fd[2], len;
    char message[CHAR_BUFSIZE];

    if(socketpair(AF_LOCAL, SOCK_STREAM, 0, fd) == -1) {
        return 1;
    }

    /* Print a message into one end of the socket */
    snprintf(message, CHAR_BUFSIZE, "First message");
    write(fd[0], message, strlen(message) + 1);

    /* Print a message into the other end of the socket */
    snprintf(message, CHAR_BUFSIZE, "Second message");
```
write(fd[1], message, strlen(message) + 1);

/* Read back the data written to the first socket */
len = read(fd[0], message, CHAR_BUFSIZE-1);
message[len] = '\0';
printf("Read [%s] from first fd \n", message);

/* Read back the data written to the second socket */
len = read(fd[1], message, CHAR_BUFSIZE-1);
message[len] = '\0';
printf("Read [%s] from second fd \n", message);

close(fd[0]);
close(fd[1]);

return 0;
}

Classification:
POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

read(), socket(), write()
Synopsis:

```c
#include <sys/select.h>

int SOCKSinit( char * proname );
```

Arguments:

`proname`  The name that you want to associate with your program. The default is `SOCKSclient`.

Library:

`libsocks`

Use the `-l socks` option to `qcc` to link against this library.

Description:

The `SOCKSinit()` function initializes some defaults for the SOCKS library and also sets the program name that appears in the `syslog` output.

You don’t have to call this function before making a SOCKS library call (but if you don’t, a generic “SOCKSclient” appears instead of the program name).

For more information about SOCKS and its libraries, see the appendix, SOCKS — A Basic Firewall.

Returns:

0  Success.

1  An error occurred (`errno` is set).

Classification:

SOCKS
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

Raccept(), Rbind(), Rconnect(), Rgetsockname(), Rlisten(), Rrcmd(), Rselect()

SOCKS — A Basic Firewall
Open a file for shared access

Synopsis:
```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <share.h>

int sopen( const char* filename,  
           int oflag,  
           int share,  
           ... ) ;
```

Arguments:

- **filename**: The path name of the file that you want to open.
- **oflag**: Flags that specify the status and access modes of the file. This argument is a combination of the following bits (defined in `<fcntl.h>`):
  - `O_RDONLY` — permit the file to be only read.
  - `O_WRONLY` — permit the file to be only written.
  - `O_RDWR` — permit the file to be both read and written.
  - `O_APPEND` — cause each record that’s written to be written at the end of the file.
  - `O_CREAT` — create the file if it doesn’t exist. This bit has no effect if the file already exists.
  - `O_TRUNC` — truncate the file to contain no data if the file exists; this bit has no effect if the file doesn’t exist.
  - `O_EXCL` — open the file for exclusive access. If the file exists and you also specify `O_CREAT`, the open fails (that is, use `O_EXCL` to ensure that the file doesn’t already exist).
- **share**: The shared access for the file. This is a combination of the following bits (defined in `<share.h>`):
sopen()

- SH_COMPAT — set compatibility mode.
- SH_DENYRW — prevent read or write access to the file.
- SH_DENYWR — prevent write access to the file.
- SH_DENYRD — prevent read access to the file.
- SH_DENYNO — permit both read and write access to the file.

If you set O_CREAT in oflag, you must also specify the following argument:

`mode` An object of type `mode_t` that specifies the access mode that you want to use for a newly created file. For more information, see “Access permissions” in the documentation for `stat()`.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sopen()` function opens a file at the operating system level for shared access. The name of the file to be opened is given by `filename`.

The file is accessed according to the access mode specified by `oflag`. You must specify O_CREAT if the file doesn’t exist.

The sharing mode of the file is given by the `share` argument. The optional argument is the file permissions to be used when O_CREAT flag is on in the `oflag` mode; you must provide this when the file is to be created.

The `sopen()` function applies the current file permission mask to the specified permissions (see `umask()`).

Note that
The `sopen()` function ignores advisory locks that you may have set by calling `fcntl()`.

**Returns:**

A descriptor for the file, or -1 if an error occurs while opening the file (`errno` is set).

**Errors:**

- **EACCES**  Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by `oflag` are denied, or the file doesn’t exist and write permission is denied for the parent directory of the file to be created.
- **EBUSY**   Sharing mode (`share`) was denied due to a conflicting open.
- **EISDIR**  The named file is a directory, and the `oflag` argument specifies write-only or read/write access.
- **ELOOP**   Too many levels of symbolic links or prefixes.
- **EMFILE**  No more descriptors available (too many open files).
- **ENOENT**  Path or file not found.
- **ENOSYS**  The `sopen()` function isn’t implemented for the filesystem specified in `path`.
Examples:

```c
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <share.h>

int main( void )
{
    int filedes;

    /* open a file for output     */
    /* replace existing file if it exists */
    filedes = sopen( "file", O_WRONLY | O_CREAT | O_TRUNC,
                     SH_DENYWR, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    /* read a file which is assumed to exist */
    filedes = sopen( "file", O_RDONLY, SH_DENYWR );

    /* append to the end of an existing file */
    /* write a new file if file doesn’t exist */
    filedes = sopen( "file", O_WRONLY | O_CREAT | O_APPEND,
                     SH_DENYWR, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );

    return EXIT_SUCCESS;
}
```

Classification:

Unix

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

continued...
See also:

chsize(), close(), creat(), dup(), dup2(), eof(), execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(), fcnt(), fileno(), fstat(), isatty(), lseek(), open(), read(), stat(), tell(), umask(), write()
Synopsis:

```c
#include <unistd.h>

int sopenfd( int fd,
             int oflag,
             int sflag );
```

Arguments:

- **fd**: A file descriptor associated with the file that you want to open.

- **oflag**: How you want to open the file; a combination of the following bits:
  - `O_RDONLY` — permit the file to be only read.
  - `O_WRONLY` — permit the file to be only written.
  - `O_RDWR` — permit the file to be both read and written.
  - `O_APPEND` — cause each record that’s written to be written at the end of the file.
  - `O_TRUNC` — if the file exists, truncate it to contain no data. This flag has no effect if the file doesn’t exist.

- **sflag**: How you want the file to be shared; a combination of the following bits:
  - `SH_COMPAT` — set compatibility mode.
  - `SH_DENYRW` — prevent read or write access to the file.
  - `SH_DENYWR` — prevent write access to the file.
  - `SH_DENYRD` — prevent read access to the file.
  - `SH_DENYNOR` — permit both read and write access to the file.
Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sopenfd()` function opens for shared access the file associated with the file descriptor, `fd`. The access mode, `oflag`, must be equal to or more restrictive than the access mode of the source `fd`.

Note that:

```c
openfd( fd, oflag );
```

is the same as:

```c
sopenfd( fd, oflag, SH_DENYNO );
```

Returns:

The file descriptor, or -1 if an error occurs (`errno` is set).

Errors:

- **EBADF** Invalid file descriptor `fd`.
- **EACCES** The access mode specified by `oflag` isn’t equal to or more restrictive than the access mode of the source `fd`.
- **EBUSY** Sharing mode (`sflag`) was denied due to a conflicting open.

Classification:

Unix
sopenfd()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

openfd()
spawn()

Create and execute a new child process

Synopsis:

```c
#include <spawn.h>

pid_t spawn( const char * path,
             int fd_count,
             const int fd_map[ ],
             const struct inheritance * inherit,
             char * const argv[ ],
             char * const envp[ ] );
```

Arguments:

- **path**: The full path name of the executable.
- **fd_count**: The number of entries in the `fd_map` array.
- **fd_map**: An array of file descriptors that you want the child process to inherit. If `fd_count` isn’t 0, `fd_map` must contain at least `fd_count` file descriptors, up to `OPEN_MAX` FDs. If `fd_count` is 0, `fd_map` is ignored.
  - If you set `fdmap[X]` to SPAWN_FDCLOSED instead of to a valid file descriptor, the file descriptor `X` is closed in the child process.
  - If `fd_count` is 0, all file descriptors (except for the ones modified with `fcntl()`’s FD_CLOEXEC flag) are inherited by the child process.
- **inherit**: A structure, of type `struct inheritance`, that indicates what you want the child process to inherit from the parent. For more information, see “inheritance structure,” below.
- **argv**: A pointer to an argument vector. The value in `argv[0]` should represent the filename of the program being loaded, but can be NULL if no arguments are being passed. The last member of `argv` must be a NULL pointer. The value of `argv` can’t be NULL.
spawn()

`envp` A pointer to an array of character pointers, each pointing to a string defining an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

```
variable=value
```

that’s used to define an environment variable. If the value of `envp` is NULL, then the child process inherits the environment of the parent process.

Library:

libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `spawn()` function creates and executes a new child process, named in `path`.

If the child process is a shell script, the first line must start with `#!/`, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The `spawn()` function is a QNX Neutrino function (based on the POSIX 1003.1d draft standard). The C library also includes several specialized `spawn*()` functions. Their names consist of `spawn` followed by several letters:

<table>
<thead>
<tr>
<th>This suffix</th>
<th>Indicates the function takes these arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>e</code></td>
<td>An array of environment variables.</td>
</tr>
</tbody>
</table>

continued...
**This suffix:** Indicates the function takes these arguments:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A NULL-terminated list of arguments to the program.</td>
</tr>
<tr>
<td>p</td>
<td>A relative path. If the path doesn’t contain a slash, the PATH environment variable is searched for the program. This suffix also lets the #! construction work; see SPAWN_CHECK_SCRIPT, below.</td>
</tr>
<tr>
<td>v</td>
<td>A vector of arguments to the program.</td>
</tr>
</tbody>
</table>

As shown below, these functions eventually call `spawn()`, which in turn sends a message to the process manager.

Most of the spawn*() functions do a lot of work before a message is sent to procnto.

The child process inherits the following attributes of the parent process:

- process group ID (unless SPAWN_SETGROUP is set in `inherit.flags`)
The child process has several differences from the parent process:

- Signals set to be caught by the parent process are set to the default action (SIG_DFL).

- The child process’s `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` are tracked separately from the parent’s.

- File locks set by the parent aren’t inherited.

- Per-process timers created by the parent aren’t inherited.

- Memory locks and mappings set by the parent aren’t inherited.

The child process also has these differences from the parent process if you haven’t set the SPAWN_EXEC flag:

- The number of seconds left until a SIGALRM signal would be generated is set to zero for the child process.

- The set of pending signals for the child process is empty.
If the child process is spawned on a remote node, the process group ID and the session membership aren’t set; the child process is put into a new session and a new process group.

The child process can access its environment by using the `environ` global variable (found in `<unistd.h>`).

If the `path` is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the child process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the child process is set to the owner ID of `path`. Similarly, if the set-group ID mode bit is set, the effective group ID of the child process is set to the group ID of `path`.

The real user ID, real group ID and supplementary group IDs of the child process remain the same as those of the parent process. The effective user ID and effective group ID of the child process are saved as the saved set-user ID and the saved set-group ID used by the `setuid()`.

A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

**inheritance structure**

The `inheritance` structure contains at least these members:

**unsigned long flags**

Zero or more of the following bits:

- `SPAWN_ALIGN_DEFAULT` — use the system’s default settings for alignment.
- `SPAWN_ALIGN_FAULT` — try to always fault data misalignment references.
- `SPAWN_ALIGN_NOFAULT` — don’t fault on misalignment; attempt to fix it (this may be slow).
SPAWN_CHECK_SCRIPT — if path starts with 
#1, spawn that binary instead, passing path as the first argument, then the arguments after the binary, then the original arguments.

SPAWN_DEBUG — debug process (this is used only for debugging the kernel itself).

SPAWN_EXEC — cause the spawn to act like exec*(): replace the calling program in memory with the newly loaded program. If successful, no return is made to the calling program.

SPAWN_EXPLICIT_CPU — set the runmask and inherit mask equal to the runmask member of the inheritance structure. If this flag isn’t set, the child inherits the inherit mask of the calling thread.

This flag was added in the QNX Neutrino Core OS 6.3.2.

SPAWN_EXPLICIT_SCHED — set the scheduling policy to the value of the policy member, and the scheduling parameters to the value of the param member.

SPAWN_HOLD — hold a process for debugging (i.e. send the SIGHOLD signal to the process before it executes its first instruction).

SPAWN_NOZOMBIE — prevent the child process from becoming a zombie on its death. No child return or exit information will be available.

SPAWN_SEARCH_PATH — search the PATH environment variable for the executable.

SPAWN_SETGROUP — set the child’s process group to the value in the pgroup member. If this flag isn’t set, the child process is part of the current process group.
• SPAWN_SETND — spawn the child process on the node specified by the nd member.

• SPAWN_SETSID — make the new process a session leader.

• SPAWN_SETSIGDEF — use the sigdefault member to specify the child process’s set of defaulted signals. If this flag isn’t specified, the child process inherits the parent process’s signal actions.

• SPAWN_SETSIGIGN — set the handling for signals defined in the sigignore member to SIG_IGN.

• SPAWN_SETSIGMASK — use the sigmask member to specify the child process’s signal mask.

• SPAWN_SETSTACKMAX — set the maximum stack size to the value of the stack_max member.

• SPAWN_TCSETPGROUP — start a new terminal group.

The <spawn.h> file also defines SPAWN_ALIGN_MASK. It’s a mask for the alignment flags listed above.

pid_t pgroup  The child process’s group if SPAWN_SETGROUP is specified in the flags member.

If SPAWN_SETGROUP is set in inherit.flags and inherit.pgroup is set to SPAWN_NEWPGROUP, the child process starts a new process group with the process group ID set to its process ID.

int runmask  A mask that indicates which processors the child process can run on.
This member was added in the QNX Neutrino Core OS 6.3.2.

**sigset_t sigmask**

The child process’s signal mask if you specify SPAWN_SETSIGMASK in the `flags` member.

**sigset_t sigdefault**

The child process’s set of defaulted signals if you specify SPAWN_SETSIGDEF in the `flags` member.

**sigset_t sigignore**

The child process’s set of ignored signals if you specify SPAWN_SETSIGIGN in the `flags` member.

**unsigned long stack_max**

The maximum stack size for the child process, if you set SPAWN_SETSTACKMAX in the `flags` member.

**int policy**

The scheduling policy for the child process, if you set SPAWN_EXPLICIT_SCHED in the `flags` member. The policy must be one of the following:

- **SCHED_FIFO** — a fixed-priority scheduler in which the highest priority ready thread runs until it blocks or is preempted by a higher priority thread.
- **SCHED_RR** — similar to SCHED_FIFO, except that threads at the same priority level timeslice (round robin) every `4 \times` the clock period (see `ClockPeriod()`).
- **SCHED_OTHER** — currently the same as SCHED_RR.
- **SCHED_SPORADIC** — sporadic scheduling.
spawn()

uint32_t nd

The node descriptor of the remote node on which to spawn the child process. This member is used only if you set SPAWN_SETND in the flags member.

If you want to spawn() remotely, set the nd member to the node descriptor. See the netmgr_strtond() function.

struct sched_param param

Scheduling parameters for the child process, if you set SPAWN_EXPLICIT_SCHED in the flags member. For more information, see the documentation for sched_param.

Returns:

The process ID of the child process, or -1 if an error occurs (errno is set).

If you set SPAWN_EXEC in the flags member of the inheritance structure, spawn() doesn’t return, unless an error occurred.

Errors:

E2BIG The number of bytes used by the argument list and environment list of the new child process is greater than ARG_MAX bytes.

EACCESS Search permission is denied for a directory listed in the path prefix of the new child process or the child process’s file doesn’t have the execute bit set or path’s filesystem was mounted with the ST_NOEXEC flag.

EAGAIN Insufficient resources available to create the child process.

EBADF An entry in fd_map refers to an invalid file descriptor, or an error occurred duplicating open file descriptors to the new process.
spawn()

EFAULT One of the buffers specified in the function call is invalid.

ELOOP Too many levels of symbolic links or prefixes.

EMFILE Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT The file identified by the path argument is empty, or one or more components of the pathname of the child process don’t exist.

ENOEXEC The child process’s file has the correct permissions, but isn’t in the correct format for an executable. (This error doesn’t occur if SPAWN_CHECK_SCRIPT is set in the flags member of the inheritance structure.)

ENOMEM Insufficient memory available to create the child process.

ENOSYS The spawn() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of the path prefix of the child process isn’t a directory.

Classification:

QNX Neutrino

Safety

Cancellation point  No

continued…
spawn()

Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

execl(), execle(), execlp(), execvp(), execve(), execvp(), execvpe(), getenv(), netmgr_strtond(), putenv(), sched_param, setenv(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), wait(), waitpid()
spawnl()

Spawn a child process, given a list of arguments

Synopsis:

```c
#include <process.h>

int spawnl( int mode,
            const char * path,
            const char * arg0,
            const char * arg1...,
            const char * argn,
            NULL );
```

Arguments:

- **mode** How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - `P_WAIT` — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - `P_NOWAIT` — execute the parent program concurrently with the new child process.
  - `P_NOWAITO` — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
  - `P_OVERLAY` — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

- **path** The full path name of the executable.

- **arg0, argn, NULL** The arguments that you want to pass to the new process. You must terminate the list with an argument of NULL.
Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnl()` function creates and executes a new child process, named in `path` with a NULL-terminated list of arguments in `arg0 ... argn`. This function calls `spawnve()`.

If the new child process is a shell script, the first line must start with `#!/`, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The `spawnl()` function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It builds an `argv[ ]` array before calling `spawn()`.

For a diagram of how the `spawn*` functions are related, see the description of `spawn()`.

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. At least one argument, `arg0`, must be passed to the child process. By convention, this first argument is a pointer to the name of the new child process.

The child process inherits the parent’s environment. The environment is the collection of environment variables whose values that have been defined with the `export` shell command, the `env` utility, or by the successful execution of the `putenv()` or `setenv()` function. A program may read these values with the `getenv()` function.
A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

**Returns:**

The `spawnl()` function’s return value depends on the `mode` argument:

<table>
<thead>
<tr>
<th><code>mode</code></th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_WAIT</td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td>P_NOWAIT</td>
<td>The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the <code>waitpid()</code> function, giving it this process ID.</td>
</tr>
<tr>
<td>P_NOWAITO</td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You can’t get the exit status of a P_NOWAITO process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (`errno` is set).

**Errors:**

- **E2BIG** The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.
- **EACCESS** Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN** Insufficient resources available to create the child process.
- **EBADF** An error occurred duplicating open file descriptors to the new process.
spawnl()

EFAULT One of the buffers specified in the function call is invalid.

ELOOP Too many levels of symbolic links or prefixes.

EMFILE Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG

The length of path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT The file identified by the path argument is empty, or one or more components of the pathname of the child process don’t exist.

ENOEXEC The child process’s file has the correct permissions, but isn’t in the correct format for an executable.

ENOMEM Insufficient memory available to create the child process.

ENOSYS The spawnl() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of the path prefix of the child process isn’t a directory.

Examples:

Run myprog as if the user had typed:

myprog ARG1 ARG2

at the command-line:

#include <stddef.h>
#include <process.h>

int exit_val;
...
exit_val = spawnl( P_WAIT, "myprog",
                "myprog", "ARG1", "ARG2", NULL );
...
The program is found if `myprog` is in the current working directory.

**Classification:**

QNX 4

**Safety**

- Cancellation point: Read the Caveats
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

**Caveats:**

If `mode` is P_WAIT, this function is a cancellation point.

**See also:**

- `execl()`, `execlp()`, `execle()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawn(,)`, `spawnvp()`, `spawnvpe()`, `wait()`, `waitpid()`
spawnle()

Spawn a child process, given a list of arguments and an environment

Synopsis:

```c
#include <process.h>

int spawnle( int mode,
             const char * path,
             const char * arg0,
             const char * arg1...
             const char * argn,
             NULL,
             const char * envp[] );
```

Arguments:

- **mode** How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - P_WAIT — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - P_NOWAIT — execute the parent program concurrently with the new child process.
  - P_NOWAITO — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
  - P_OVERLAY — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

- **path** The full path name of the executable.

- **arg0, argn, NULL**
  The arguments that you want to pass to the new process. You must terminate the list with an argument of NULL.

- **envp** NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable.
The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that's used to define an environment variable.

Library:

<table>
<thead>
<tr>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc</td>
</tr>
</tbody>
</table>

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnle()` function creates and executes a new child process, named in `path` with NULL-terminated list of arguments in `arg0 ... argn` and with the environment specified in `envp`. This function calls `spawnve()`.

If the new child process is a shell script, the first line must start with `#!`, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The `spawnle()` function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It builds `argv[]` and `envp[]` arrays before calling `spawn()`.

For a diagram of how the `spawn*` functions are related, see the description of `spawn()`.

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. At least one argument, `arg0`, must be passed to the child process. By convention, this first argument is a pointer to the name of the new child process.
If `envp` is NULL, the child process inherits the environment of the parent process. The new process can access its environment by using the `environ` global variable (found in `<unistd.h>`).

A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

**Returns:**

The `spawnle()` function’s return value depends on the `mode` argument:

<table>
<thead>
<tr>
<th><code>mode</code></th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_WAIT</td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td>P_NOWAIT</td>
<td>The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the <code>waitpid()</code> function, giving it this process ID.</td>
</tr>
<tr>
<td>P_NOWAITO</td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You can’t get the exit status of a P_NOWAITO process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (`errno` is set).

**Errors:**

- **E2BIG** The number of bytes used by the argument list or environment list of the new child process is greater than `ARG_MAX` bytes.
- **EACCESS** Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN** Insufficient resources available to create the child process.
- **EBADF** An error occurred duplicating open file descriptors to the new process.
spawnle()

EFAULT One of the buffers specified in the function call is invalid.

ELOOP Too many levels of symbolic links or prefixes.

EMFILE Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG
The length of path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT The file identified by the path argument is empty, or one or more components of the pathname of the child process don’t exist.

ENOEXEC The child process’s file has the correct permissions, but isn’t in the correct format for an executable.

ENOMEM Insufficient memory available to create the child process.

ENOSYS The spawnle() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of the path prefix of the child process isn’t a directory.

Examples:

Run myprog as if the user had typed:

myprog ARG1 ARG2

at the command-line:

#include <stddef.h>
#include <process.h>

char *env_list[] = { "SOURCE=MYDATA",
                    "TARGET=OUTPUT",};
spawnle()

```

    "lines=65",
    NULL

spawnle( P_WAIT, "myprog",
    "myprog", "ARG1", "ARG2", NULL,
    env_list );
```

The program is found if `myprog` is in the current working directory.
The environment for the child program consists of the three
environment variables `SOURCE`, `TARGET` and `lines`.

**Classification:**

QNX 4

**Safety**

- Cancellation point: Read the *Caveats*
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

**Caveats:**

If `mode` is `P_WAIT`, this function is a cancellation point.

**See also:**

- `execl()`, `execle()`, `exechp()`, `exechpe()`, `execv()`, `execve()`, `execvp()`,
- `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnlp()`,
- `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`,
- `wait()`, `waitpid()`
spawnlp()

Spawn a child process, given a list of arguments and a relative path

Synopsis:

```
#include <process.h>

int spawnlp( int mode,
            const char * file,
            const char * arg0,
            const char * arg1,...,
            const char * argn,
            NULL );
```

Arguments:

- **mode**  How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - P_WAIT — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - P_NOWAIT — execute the parent program concurrently with the new child process.
  - P_NOWAITO — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
  - P_OVERLAY — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec`*() function.

- **file**  The name of the executable file. If this argument contains a slash, it’s used as the pathname of the executable; otherwise, the function searches for file in the directories listed in the PATH environment variable.

- **arg0, argn, NULL**  The arguments that you want to pass to the new process. You must terminate the list with an argument of NULL.
Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The spawnlp() function creates and executes a new child process, named in file with NULL-terminated list of arguments in arg0 ... argn. This function calls spawnvpe().

If the new child process is a shell script, the first line must start with #!, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The spawnlp() function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It builds an argv[ ] array before calling spawnp().

For a diagram of how the spawn* functions are related, see the description of spawn().

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. At least one argument, arg0, must be passed to the child process. By convention, this first argument is a pointer to the name of the new child process.

The child process inherits the parent’s environment. The environment is the collection of environment variables whose values that have been defined with the export shell command, the env utility, or by the successful execution of the putenv() or setenv() function. A program may read these values with the getenv() function.
A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

Returns:

The `spawnlp()` function’s return value depends on the `mode` argument:

<table>
<thead>
<tr>
<th><code>mode</code></th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_WAIT</td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td>P_NOWAIT</td>
<td>The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the <code>waitpid()</code> function, giving it this process ID.</td>
</tr>
<tr>
<td>P_NOWAITO</td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You can’t get the exit status of a P_NOWAITO process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (`errno` is set).

Errors:

- **E2BIG**: The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.
- **EACCESS**: Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN**: Insufficient resources available to create the child process.
- **EBADF**: An error occurred duplicating open file descriptors to the new process.
spawnlp() © 2007, QNX Software Systems GmbH & Co. KG.

EFAULT One of the buffers specified in the function call is invalid.

ELOOP Too many levels of symbolic links or prefixes.

EMFILE Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

ENAMETOOLONG The length of file exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENOENT The file identified by the file argument is empty, or one or more components of the pathname of the child process don’t exist.

ENOEXEC The child process’s file has the correct permissions, but isn’t in the correct format for an executable.

ENOMEM Insufficient memory available to create the child process.

ENOSYS The spawnlp() function isn’t implemented for the filesystem specified in file.

ENOTDIR A component of the path prefix of the child process isn’t a directory.

**Classification:**

QNX 4

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Read the Caveats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

*continued...*
spawnlp()

Safety

| Thread | Yes |

Caveats:

If `mode` is P_WAIT, this function is a cancellation point.

See also:

`execl()`, `execle()`, `exclp()`, `exclpe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnvp()`, `spawnvpe()`, `wait()`, `waitpid()`
Spawn a child process, given a list of arguments, an environment, and a relative path

Synopsis:

```c
#include <process.h>

int spawnlpe( int mode, 
              const char * file, 
              const char * arg0, 
              const char * arg1..., 
              const char * argn, 
              NULL, 
              const char * envp[] );
```

Arguments:

- **mode**
  How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - P_WAIT — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - P_NOWAIT — execute the parent program concurrently with the new child process.
  - P_NOWAITO — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
  - P_OVERLAY — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

- **file**
  The name of the executable file. If this argument contains a slash, it’s used as the pathname of the executable; otherwise, the function searches for `file` in the directories listed in the `PATH` environment variable.

- **arg0, argn, NULL**
  The arguments that you want to pass to the new process. You must terminate the list with an argument of NULL.
spawnlpe()?

envp

NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

variable=value

that’s used to define an environment variable.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The spawnlpe() function creates and executes a new child process, named in file with NULL-terminated list of arguments in arg0 ... argn and with the environment specified in envp. This function calls spawnvpe().

If the new child process is a shell script, the first line must start with #!, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The spawnlpe() function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It builds argv[ ] and envp[ ] arrays before calling spawnp().

For a diagram of how the spawn* functions are related, see the description of spawn().

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. At least one argument, arg0, must be passed to the child process. By convention, this first argument is a pointer to the name of the new child process.
If the value of `envp` is NULL, then the child process inherits the environment of the parent process. The new process can access its environment by using the `environ` global variable (found in `<unistd.h>`).

A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

**Returns:**

The `spawnlpe()` function’s return value depends on the `mode` argument:

<table>
<thead>
<tr>
<th><code>mode</code></th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>P_WAIT</code></td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td><code>P_NOWAIT</code></td>
<td>The process ID of the child process. To get the exit status for a <code>P_NOWAIT</code> process, you must use the <code>waitpid()</code> function, giving it this process ID.</td>
</tr>
<tr>
<td><code>P_NOWAITO</code></td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You can’t get the exit status of a <code>P_NOWAITO</code> process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (`errno` is set).

**Errors:**

- **E2BIG** The number of bytes used by the argument list or environment list of the new child process is greater than `ARG_MAX` bytes.
- **EACCESS** Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN** Insufficient resources available to create the child process.
spawnlpe()

EBADF     An error occurred duplicating open file descriptors to the new process.
EFAULT    One of the buffers specified in the function call is invalid.
ELOOP      Too many levels of symbolic links or prefixes.
EMFILE     Insufficient resources available to load the new executable image or to remap file descriptors in the child process.
ENAMETOOLONG
            The length of file exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
ENOENT     The file identified by the file argument is empty, or one or more components of the pathname of the child process don’t exist.
ENOEXEC    The child process’s file has the correct permissions, but isn’t in the correct format for an executable.
ENOMEM     Insufficient memory available to create the child process.
ENOSYS     The spawnlpe() function isn’t implemented for the filesystem specified in file.
ENOTDIR    A component of the path prefix of the child process isn’t a directory.

Classification:

QNX 4

Safety

Cancellation point  Read the Caveats

continued...
spawnlpe()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

If `mode` is `P_WAIT`, this function is a cancellation point.

**See also:**

`exec()` , `execl()` , `execlp()` , `execle()` , `execv()` , `execve()` , `execvp()` , `execvpe()` , `getenv()` , `putenv()` , `setenv()` , `spawn()` , `spawnl()` , `spawnlpe()` , `spawnlp()` , `spawnp()` , `spawnv()` , `spawnve()` , `spawnvp()` , `spawnvpe()` , `wait()` , `waitpid()`
/*
 * spawnp()
 * Create and execute a new child process, given a relative path
 *
 * Synopsis:
 * 
 * #include <spawn.h>
 *
 * pid_t spawnp( const char * file,
 *     int fd_count,
 *     const int fd_map[ ],
 *     const struct inheritance * inherit,
 *     char * const argv[ ],
 *     char * const envp[ ]);
 *
 * Arguments:
 *
 * file
 * If this argument contains a slash, it’s used as the
 * pathname of the executable; otherwise, the PATH
 * environment variable is searched for file.
 *
 * fd_count
 * The number of entries in the fd_map array.
 *
 * fd_map
 * An array of file descriptors that you want the child
 * process to inherit. If fd_count isn’t 0, fd_map must
 * contain at least fd_count file descriptors, up to
 * OPEN_MAX FDs. If fd_count is 0, fd_map is ignored.
 *
 * If you set fdmap[X] to SPAWN_FDCLOSED instead of
 * to a valid file descriptor, the file descriptor X is closed in
 * the child process.
 *
 * If fd_count is 0, all file descriptors (except for the ones
 * modified with fcntl()’s FD_CLOEXEC flag) are inherited
 * by the child process.
 *
 * inherit
 * A structure, of type struct inheritance, that
 * indicates what you want the child process to inherit from
 * the parent. For more information, see “inheritance
 * structure,” in the documentation for spawn().
 */
If you want to `spawnp()` remotely, set the `nd` member of the inheritance structure to the node descriptor. See the `netmgr_strtond()` function.

**argv**  
A pointer to an argument vector. The value in `argv[0]` should represent the filename of the program being loaded, but can be NULL if no arguments are being passed. The last member of `argv` must be a NULL pointer. The value of `argv` can't be NULL.

**envp**  
A pointer to an array of character pointers, each pointing to a string defining an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

```
variable=value
```

that's used to define an environment variable. If the value of `envp` is NULL, then the child process inherits the environment of the parent process.

**Library:**

** libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `spawnp()` function creates and executes a new child process, named in `file`. It sets the SPAWN_CHECK_SCRIPT and SPAWN_SEARCH_PATH flags, and then calls `spawn()`.
If the new child process is a shell script, the first line must start with 
#, followed by the path and arguments of the shell to be run to 
interpret the script. The script must also be marked as executable.

The spawnp() function is a QNX function (based on the POSIX 
1003.1d draft standard). The C library also includes several 
specialized spawnp() functions. Their names consist of spawnp 
followed by several letters:

<table>
<thead>
<tr>
<th>This suffix:</th>
<th>Indicates the function takes these arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>An array of environment variables.</td>
</tr>
<tr>
<td>l</td>
<td>A NULL-terminated list of arguments to the program.</td>
</tr>
<tr>
<td>p</td>
<td>A relative path. If the path doesn’t contain a slash, the PATH environment variable is searched for the program. This suffix also lets the #! construction work; see SPAWN_CHECK_SCRIPT in the documentation for spawnp().</td>
</tr>
<tr>
<td>v</td>
<td>A vector of arguments to the program.</td>
</tr>
</tbody>
</table>

For a diagram of how the spawnp* functions are related, see the description of spawnp().

The child process inherits the following attributes of the parent process:

- process group ID (unless SPAWN_SETGROUP is set in inherit.flags)
- session membership
- real user ID and real group ID
• effective user ID and effective group ID
• supplementary group IDs
• priority and scheduling policy
• current working directory and root directory
• file-creation mask
• signal mask (unless SPAWN_SETSIGMASK is set in inherit.flags)
• signal actions specified as SIG_DFL
• signal actions specified as SIG_IGN (except the ones modified by inherit.sigdefault when SPAWN_SETSIGDEF is set in inherit.flags).

The child process has several differences from the parent process:

• Signals set to be caught by the parent process are set to the default action (SIG_DFL).
• The child process’s tms_utime, tms_stime, tms_cutime, and tms_cstime are tracked separately from the parent’s.
• The number of seconds left until a SIGALRM signal would be generated is set to zero for the child process.
• The set of pending signals for the child process is empty.
• File locks set by the parent aren’t inherited.
• Per-process timers created by the parent aren’t inherited.
• Memory locks and mappings set by the parent aren’t inherited.

If the child process is spawned on a remote node, the process group ID and the session membership aren’t set; the child process is put into a new session and a new process group.

The child process can access its environment by using the environ global variable (found in <unistd.h>).
If the file is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the child process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the child process is set to the owner ID of file. Similarly, if the set-group ID mode bit is set, the effective group ID of the child process is set to the group ID of file. The real user ID, real group ID and supplementary group IDs of the child process remain the same as those of the parent process. The effective user ID and effective group ID of the child process are saved as the saved set-user ID and the saved set-group ID used by the setuid() function.

The following arguments are passed to the underlying call (spawn()):

\[
\text{spawnp(file, fd_count, fd_map, inherit, argv, envp)}
\]

- Create an empty attribute if inherit is NULL
- Set the CHECK_SCRIPT flag
- Set the SPAWN_SEARCH_PATH
- Call \text{spawn(file, fd_count, fd_map, inherit/attr, argv, envp)}.

A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

**Returns:**

The process ID of the child process, or -1 if an error occurs (errno is set).

If you set SPAWN_EXEC in the flags member of the inheritance structure, spawnp() doesn’t return, unless an error occurred.
### Errors:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2BIG</td>
<td>The number of bytes used by the argument list and environment list of the new child process is greater than ARG_MAX bytes.</td>
</tr>
<tr>
<td>EACCESS</td>
<td>Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set or file’s filesystem was mounted with the ST_NOEXEC flag.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Insufficient resources available to create the child process.</td>
</tr>
<tr>
<td>EBADF</td>
<td>An entry in fd_map refers to an invalid file descriptor, or an error occurred duplicating open file descriptors to the new process.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>One of the buffers specified in the function call is invalid.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many levels of symbolic links or prefixes.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Insufficient resources available to load the new executable image or to remap file descriptors in the child process.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>The length of file exceeds PATH_MAX or a pathname component is longer than NAME_MAX.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The file identified by the file argument is empty, or one or more components of the pathname of the child process don’t exist.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>The child process’s file has the correct permissions, but isn’t in the correct format for an executable.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory available to create the child process.</td>
</tr>
</tbody>
</table>
The `spawnp()` function isn’t implemented for the filesystem specified in `file`.

A component of the path prefix of the child process isn’t a directory.

**Classification:**

| QNX Neutrino |

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`exect(), execle(), execlpe(), execv(), execve(), execvp(), execvpe(), getenv(), netmgr_strtond(), putenv(), setenv(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnv(), spawnve(), spawnvp(), spawnvpe(), wait(), waitpid()`
spawnv()

Spell a child process, given a vector of arguments

Synopsis:

```c
#include <process.h>

int spawnv( int mode,
            const char * path,
            char * const argv[] );
```

Arguments:

- **mode** How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - P_WAIT — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - P_NOWAIT — execute the parent program concurrently with the new child process.
  - P_NOWAITO — execute the parent program concurrently with the new child process. You can’t use wait() to obtain the exit code.
  - P_OVERLAY — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate exec*() function.

- **path** The full path name of the executable.

- **argv** A pointer to an argument vector. The value in argv[0] should represent the filename of the program that you’re loading. The last member of argv must be a NULL pointer. The value of argv can’t be NULL, and argv[0] can’t be a NULL pointer, even if you’re not passing any argument strings.
spawnv()

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The spawnv() function creates and executes a new child process, named in path with the NULL-terminated list of arguments in the argv vector. This function calls spawnvpe().

If the new child process is a shell script, the first line must start with #!, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The spawnv() function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It calls spawnve() before calling spawn().

For a diagram of how the spawn* functions are related, see the description of spawn().

The child process inherits the parent’s environment. The environment is the collection of environment variables whose values that have been defined with the export shell command, the env utility, or by the successful execution of the putenv() or setenv() function. A program may read these values with the getenv() function.

A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

Returns:

The spawnv() function’s return value depends on the mode argument:
### spawnv()

#### mode | Return value
--- | ---
P_WAIT | The exit status of the child process.
P_NOWAIT | The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the `waitpid()` function, giving it this process ID.
P_NOWAITO | The process ID of the child process, or 0 if the process is being started on a remote node. You can’t get the exit status of a P_NOWAITO process.

If an error occurs, -1 is returned (`errno` is set).

#### Errors:

- **E2BIG** The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.
- **EACCESS** Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN** Insufficient resources available to create the child process.
- **EBADF** An error occurred duplicating open file descriptors to the new process.
- **EFAULT** One of the buffers specified in the function call is invalid.
- **EINVAL** Too many levels of symbolic links or prefixes.
- **EMFILE** Insufficient resources available to load the new executable image or to remap file descriptors in the child process.
- **ENAMETOOLONG** The length of `path` exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
spawnv()

ENOENT  The file identified by the path argument is empty, or one or more components of the pathname of the child process don’t exist.

ENOEXEC  The child process’s file has the correct permissions, but isn’t in the correct format for an executable.

ENOMEM  Insufficient memory available to create the child process.

ENOSYS  The spawnv() function isn’t implemented for the filesystem specified in path.

ENOTDIR  A component of the path prefix of the child process isn’t a directory.

Examples:

Run myprog as if a user had typed:

myprog ARG1 ARG2

at the command-line:

#include <stddef.h>
#include <process.h>

char *arg_list[] = { "myprog", "ARG1", "ARG2", NULL };  
spawvn( P_WAIT, "myprog", arg_list );

The program is found if myprog is in the current working directory.

Classification:

QNX 4

Safety

Cancellation point  Read the Caveats

continued...
**spawnv()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

If *mode* is P_WAIT, this function is a cancellation point.

**See also:**

`execl()`, `execl()`, `exectl()`, `execlp()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnn()`, `spawnpe()`, `spawnvp()`, `spawnvpe()`, `wait()`, `waitpid()`
Spawn a child process, given a vector of arguments and an environment

Synopsis:

```c
#include <process.h>

int spawnve( int mode,
            const char * path,
            char * const argv[],
            char * const envp[] );
```

Arguments:

- **mode** How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - P_WAIT — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - P_NOWAIT — execute the parent program concurrently with the new child process.
  - P_NOWAITO — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
  - P_OVERLAY — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

- **path** The full path name of the executable.

- **argv** A pointer to an argument vector. The value in `argv[0]` should represent the filename of the program that you’re loading. The last member of `argv` must be a NULL pointer. The value of `argv` can’t be NULL, and `argv[0]` can’t be a NULL pointer, even if you’re not passing any argument strings.

- **envp** NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable.
The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

\[ \text{variable} = \text{value} \]

that’s used to define an environment variable.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `spawnve()` function creates and executes a new child process, named in `path` with the NULL-terminated list of arguments in the `argv` vector.

If the new child process is a shell script, the first line must start with `#!/`, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The `spawnve()` function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It calls `spawn()`.

For a diagram of how the `spawn*` functions are related, see the description of `spawn()`.

If the value of `envp` is NULL, then the child process inherits the environment of the parent process. The new process can access its environment by using the `environ` global variable (found in `<unistd.h>`).

The following arguments are passed to the underlying call (`spawn()`):

\[ \text{spawnve} (\text{mode}, \text{path}, \text{argv}, \text{envp}) \]

- Create temporary attribute structure and set the flags member based on the `mode`: 
A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

Returns:

The `spawnve()` function’s return value depends on the `mode` argument:

<table>
<thead>
<tr>
<th>mode</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_WAIT</td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td>P_NOWAIT</td>
<td>The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the <code>waitpid()</code> function, giving it this process ID.</td>
</tr>
<tr>
<td>P_NOWAITO</td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You can’t get the exit status of a P_NOWAITO process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (`errno` is set).

Errors:

- **E2BIG** The number of bytes used by the argument list or environment list of the new child process is greater than ARG_MAX bytes.
- **EACCESS** Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN** Insufficient resources available to create the child process.
spawnve()

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>An error occurred duplicating open file descriptors to the new process.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>One of the buffers specified in the function call is invalid.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many levels of symbolic links or prefixes.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Insufficient resources available to load the new executable image or to remap file descriptors in the child process.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>The length of path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The file identified by the path argument is empty, or one or more components of the pathname of the child process don’t exist.</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>The child process’s file has the correct permissions, but isn’t in the correct format for an executable.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient memory available to create the child process.</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>The spawnve() function isn’t implemented for the filesystem specified in path.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the path prefix of the child process isn’t a directory.</td>
</tr>
</tbody>
</table>

**Classification:**

QNX 4

**Safety**

Cancellation point  Read the Caveats

continued...
spawnve()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

If `mode` is P_WAIT, this function is a cancellation point.

See also:

`exec1()`, `execle()`, `execlp()`, `execle()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnvp()`, `spawnvpe()`, `wait()`, `waitpid()`
spawnvp()

Spawn a child process, given a vector of arguments and a relative path

Synopsis:

```c
#include <process.h>

int spawnvp( int mode,
              const char * file,
              char * const argv[] );
```

Arguments:

- **mode** How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
  - P_WAIT — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
  - P_NOWAIT — execute the parent program concurrently with the new child process.
  - P_NOWAITO — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
  - P_OVERLAY — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

- **file** The name of the executable file. If this argument contains a slash, it’s used as the pathname of the executable; otherwise, the function searches for `file` in the directories listed in the PATH environment variable.

- **argv** A pointer to an argument vector. The value in `argv[0]` should represent the filename of the program that you’re loading. The last member of `argv` must be a NULL pointer. The value of `argv` can’t be NULL, and `argv[0]` can’t be a NULL pointer, even if you’re not passing any argument strings.
Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `spawnvp()` function creates and executes a new child process, named in `file` with the NULL-terminated list of arguments in the `argv` vector. This function calls `spawnvpe()`.

If the new process is a shell script, the first line must start with `#!`, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The `spawnvp()` function isn’t a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It calls `spawnvpe()` before calling `spawnp()`.

For a diagram of how the `spawn*` functions are related, see the description of `spawn()`.

The child process inherits the parent’s environment. The environment is the collection of environment variables whose values that have been defined with the `export` shell command, the `env` utility, or by the successful execution of the `putenv()` or `setenv()` function. A program may read these values with the `getenv()` function.

A parent/child relationship doesn’t imply that the child process dies when the parent process dies.

Returns:

The `spawnvp()` function’s return value depends on the `mode` argument:
spawnvp()

<table>
<thead>
<tr>
<th>mode</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_WAIT</td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td>P_NOWAIT</td>
<td>The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() function, giving it this process ID.</td>
</tr>
<tr>
<td>P_NOWAITO</td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You cannot get the exit status of a P_NOWAITO process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (errno is set).

Errors:

- **E2BIG** The number of bytes used by the argument list of the new child process is greater than ARG_MAX bytes.
- **EACCESS** Search permission is denied for a directory listed in the path prefix of the new child process or the new child process’s file doesn’t have the execute bit set.
- **EAGAIN** Insufficient resources available to create the child process.
- **EBADF** An error occurred duplicating open file descriptors to the new process.
- **EFAULT** One of the buffers specified in the function call is invalid.
- **ELOOP** Too many levels of symbolic links or prefixes.
- **EMFILE** Insufficient resources available to load the new executable image or to remap file descriptors in the child process.
- **ENAMETOOLONG** The length of file and its path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
ENOENT  The file identified by the file argument is empty, or one or more components of the pathname of the new process don’t exist.

ENOEXEC  The child process file has the correct permissions, but isn’t in the correct format for an executable.

ENOMEM  Insufficient memory available to create the child process.

ENOSYS  The spawnvp() function isn’t implemented for the filesystem specified in file.

ENOTDIR  A component of the path prefix of the child process isn’t a directory.

Classification:
QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Read the Caveats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:
If mode is P_WAIT, this function is a cancellation point.

See also:

execl(), execle(), execclp(), execlpe(), execv(), execve(), execvp(), execvpe(), getenv(), putenv(), setenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnvp(), spawnve(), spawnvpe(), wait(), waitpid()
spawnvpe()

© 2007, QNX Software Systems GmbH & Co. KG.

Spawn a child process, given a vector of arguments, an environment, and a relative path

Synopsis:

```
#include <spawn.h>

int spawnvpe( int mode,
              const char * file,
              char * const argv[],
              char * const envp[] );
```

Arguments:

- **mode**
  - How you want to load the child process, and how you want the parent program to behave after the child program is initiated:
    - `P_WAIT` — load the child program into available memory, execute it, and make the parent program resume execution after the child process ends.
    - `P_NOWAIT` — execute the parent program concurrently with the new child process.
    - `P_NOWAITO` — execute the parent program concurrently with the new child process. You can’t use `wait()` to obtain the exit code.
    - `P_OVERLAY` — replace the parent program with the child program in memory and execute the child. No return is made to the parent program. This is equivalent to calling the appropriate `exec*()` function.

- **file**
  - The name of the executable file. If this argument contains a slash, it’s used as the pathname of the executable; otherwise, the function searches for `file` in the directories listed in the `PATH` environment variable.

- **argv**
  - A pointer to an argument vector. The value in `argv[0]` should represent the filename of the program that you’re loading. The last member of `argv` must be a NULL pointer. The value of `argv` can’t be NULL, and `argv[0]` can’t be a NULL pointer, even if you’re not passing any argument strings.
**spawnvpe()**

`envp` NULL, or a pointer to an array of character pointers, each pointing to a string that defines an environment variable. The array is terminated with a NULL pointer. Each pointer points to a character string of the form:

```
variable=value
```

that’s used to define an environment variable.

**Library:**

**libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `spawnvpe()` function creates and executes a new child process, named in `file` with the NULL-terminated list of arguments in the `argv` vector.

If the new child process is a shell script, the first line must start with `#!`, followed by the path and arguments of the shell to be run to interpret the script. The script must also be marked as executable.

The `spawnvpe()` function *isn’t* a POSIX 1003.1 function, and isn’t guaranteed to behave the same on all operating systems. It calls `spawnp()`.

For a diagram of how the `spawn` functions are related, see the description of `spawn()`.

If the value of `envp` is NULL, then the child process inherits the environment of the parent process. The new process can access its environment by using the `environ` global variable (found in `<unistd.h>`).

The following arguments are passed to the underlying call (`spawnp()`):
spawnvpe()

spawnvpe(mode, file, argv, envp)

- Map attribute flags.
  Create temporary attribute structure and set the flags
  member based on the mode:
  - 0 if mode = P_WAIT/P_NOWAIT
  - SPAWN_EXEC if mode = P_OVERLAY
  - SPAWN_NOZOMBIE if mode = P_NOWAITO
- Call spawnp(file, 0, 0, attr, argv, envp).

A parent/child relationship doesn’t imply that the child process dies
when the parent process dies.

Returns:

The spawnvpe() function’s return value depends on the mode
argument:

<table>
<thead>
<tr>
<th>mode</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_WAIT</td>
<td>The exit status of the child process.</td>
</tr>
<tr>
<td>P_NOWAIT</td>
<td>The process ID of the child process. To get the exit status for a P_NOWAIT process, you must use the waitpid() function, giving it this process ID.</td>
</tr>
<tr>
<td>P_NOWAITO</td>
<td>The process ID of the child process, or 0 if the process is being started on a remote node. You cannot get the exit status of a P_NOWAITO process.</td>
</tr>
</tbody>
</table>

If an error occurs, -1 is returned (errno is set).

Errors:

E2BIG The number of bytes used by the argument list or
environment list of the new child process is greater
than ARG_MAX bytes.
**spawnvpe()**

- **EACCESS**  
  Search permission is denied for a directory listed in the path prefix of the new child process or the new child process file doesn’t have the execute bit set.

- **EAGAIN**  
  Insufficient resources available to create the child process.

- **EBADF**  
  An error occurred duplicating open file descriptors to the new process.

- **EFAULT**  
  One of the buffers specified in the function call is invalid.

- **ELOOP**  
  Too many levels of symbolic links or prefixes.

- **EMFILE**  
  Insufficient resources available to load the new executable image or to remap file descriptors in the child process.

- **ENAMETOOLONG**  
  The length of file plus its path exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

- **ENOENT**  
  The file identified by the file argument is empty, or one or more components of the pathname of the child process don’t exist.

- **ENOEXEC**  
  The child process file has the correct permissions, but isn’t in the correct format for an executable.

- **ENOMEM**  
  Insufficient memory available to create the child process.

- **ENOSYS**  
  The spawnvpe() function isn’t implemented for the filesystem specified in file.

- **ENOTDIR**  
  A component of the path prefix of the child process isn’t a directory.
spawnvpe()

Classification:

QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Read the Caveats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

If mode is P_WAIT, this function is a cancellation point.

See also:

execl(), execlp(), execle(), execvp(), execve(), execvp(), execvpe(), getenv(), putenv(), setenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawmve(), spawnp(), wait(), waitpid()
Synopsis:

```c
#include <stdio.h>

int sprintf( char* buf, 
             const char* format,
             ... );
```

Arguments:
- `buf` A pointer to the buffer where you want to function to store the formatted string.
- `format` A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sprintf()` function is similar to `fprintf()`, except that `sprintf()` places the generated output into the character array pointed to by `buf`, instead of writing it to a file. A null character is placed at the end of the generated character string.

Returns:

The number of characters written into the array, not counting the terminating null character. An error can occur while converting a value for output. When an error occurs, `errno` indicates the type of error detected.
**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

/* Create temporary file names using a counter */

char namebuf[13];
int TempCount = 0;

char* make_temp_name()
{
    sprintf( namebuf, "ZZ%.6o.TMP", TempCount++ );
    return( namebuf );
}

int main( void )
{
    FILE* tf1,* tf2;

    tf1 = fopen( make_temp_name(), "w" );
    tf2 = fopen( make_temp_name(), "w" );
    fputs( "temp file 1", tf1 );
    fputs( "temp file 2", tf2 );
    fclose( tf1 );
    fclose( tf2 );
    return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`errno`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `swprintf()`, `vfprintf()`, `vfwprintf()`, `vprintf()`, `vsprintf()`, `vswprintf()`, `vwprintf()`
sqrt(), sqrtf()

Calculate the nonnegative square root of a number

Synopsis:

```c
#include <math.h>

double sqrt( double x );

float sqrtf( float x );
```

Arguments:

- `x` The number that you want to calculate the square root of.

Library:

- `libm`

Use the `-l m` option to `qcc` to link against this library.

Description:

These functions compute the nonnegative square root of `x`. A domain error occurs if the argument is negative.

Returns:

The nonnegative square root of the given number.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Examples:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main( void )
{
    printf( "%.5f\n", sqrt(.5) );
    return EXIT_SUCCESS;
}
```
The `sqrt()` and `sqrtf()` functions produce the output:

\[ 0.707107 \]

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `exp()`, `log()`, `pow()`
srand()  

Start a new sequence of pseudo-random integers

Synopsis:

```c
#include <stdlib.h>

void srand( unsigned int seed );
```

Arguments:

- **seed**: The seed of the sequence of pseudo-random integers.

Library:

- **libc**
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `srand()` function uses the argument `seed` to start a new sequence of pseudo-random integers to be returned by subsequent calls to `rand()`. A particular sequence of pseudo-random integers can be repeated by calling `srand()` with the same `seed` value. The default sequence of pseudo-random integers is selected with a `seed` value of 1.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    int i;

    srand( 982 );
    for( i = 1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }

    /* Start the same sequence over again. */

    srand( 982 );
    for( i = 1; i < 10; ++i ) {
        printf( "%d\n", rand() );
    }
}
```
srand()

}  

/*
   Use the current time as a seed to
   get a different sequence.
*/

srand( (int) time( NULL ) );
for( i = 1; i < 10; ++i ) {
    printf( "%d\n", rand() );
}

return EXIT_SUCCESS;

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

rand()
**srand48()**

Initialize a sequence of pseudo-random numbers

**Synopsis:**

```c
#include <stdlib.h>

void srand48( long seed );
```

**Arguments:**

- `seed` The seed of the sequence of pseudo-random integers.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `srand48()` is used to initialize the internal buffer `r(n)` of `drand48()`, `lrand48()`, and `mrand48()` such that the 32 bits of the seed value are copied into the upper 32 bits of `r(n)`, with the lower 16 bits of `r(n)` arbitrarily being set to `0x330E`. Additionally, the constant multiplicand and addend of the algorithm are reset to the default values: the multiplicand `a = 0xFDEECE66D = 25214903917` and the addend `c = 0xB = 11`.

**Classification:**

POSIX 1003.1 XSI

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `mrand48()`,
`nrand48()`, `seed48()`
**srandom()**  © 2007, QNX Software Systems GmbH & Co. KG.

Set the seed for a pseudo-random number generator

**Synopsis:**

```c
#include <stdlib.h>

void srandom( unsigned int seed );
```

**Arguments:**

*seed*  
The seed of the sequence of pseudo-random integers.

**Library:**

*libc*

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

---

This function is in *libc.a*, but not in *libc.so* (in order to save space).

**Description:**

The `srandom()` function initializes the current state array using the value of `seed`.

Use this function in conjunction with the following:

- `initstate()`  
  Initialize the state of the pseudo-random number generator.

- `random()`  
  Generate a pseudo-random number using a default state.

- `setstate()`  
  Specify the state of the pseudo-random number generator.

The `random()` and `srandom()` functions have (almost) the same calling sequence and initialization properties as `rand()` and `srand()`. Unlike `srand()`, `srandom()` doesn’t return the old seed because the amount of
state information used is much more than a single word. The \textit{initstate()} and \textit{setstate()} routines are provided to deal with restarting/changing random number generators. With 256 bytes of state information, the period of the random-number generator is greater than 269.

Like \textit{rand()}, \textit{random()} produces by default a sequence of numbers that can be duplicated by calling \textit{srandom()} with 1 as the seed.

After initialization, a state array can be restarted at a different point in one of two ways:

- The \textit{initstate()} function can be used, with the desired seed, state array, and size of the array.

- The \textit{setstate()} function, with the desired state, can be used, followed by \textit{srandom()} with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

\textbf{Classification:}

POSIX 1003.1 XSI

\begin{tabular}{|l|c|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & No \\
\hline
\end{tabular}

\textbf{See also:}

\textit{drand48()}, \textit{initstate()}, \textit{rand()}, \textit{random()}, \textit{setstate()}, \textit{srand()}

\begin{flushleft}
\textbf{S} \textbf{3185}
\end{flushleft}
srealloc()© 2007, QNX Software Systems GmbH & Co. KG.

Allocate, reallocate or free a block of memory

Synopsis:
#include <malloc.h>

void *srealloc( void* ptr,
                  size_t old_size,
                  size_t new_size );

Arguments:

ptr NULL, or a pointer to the block of memory that you
      want to reallocate.

old_size The current size of the block, in bytes.

new_size The size of the block to allocate, in bytes.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

This function is in libc.a, but not in libc.so (in order to save
space).

Description:

When the value of the ptr argument is NULL, a new block of memory
of new_size bytes is allocated.

If the value of new_size is zero, the corresponding _sfree() function is
called to release old_size bytes of memory memory pointed to by ptr.

Otherwise, the _srealloc() function reallocates space for an object of
new_size bytes by doing one of the following:

- Shrinking the allocated size of the allocated memory block ptr
  when new_size is sufficiently smaller than old_size.

  Or:
Extending the allocated size of the allocated memory block \texttt{ptr} if there is a large enough block of unallocated memory immediately following \texttt{ptr}.

Or:

- Allocating a new block, and copying the contents of \texttt{ptr} to the new block.

Because it’s possible that a new block will be allocated, don’t maintain any pointers into the old memory after a successful call to this function. These pointers will point to freed memory, with possible disastrous results when a new block is allocated.

The function returns NULL when the memory pointed to by \texttt{ptr} can’t be reallocated. In this case, the memory pointed to by \texttt{ptr} isn’t freed, so be sure to keep a pointer to the old memory block.

\begin{verbatim}
buffer = (char *) __srealloc( buffer, 100, 200 );
\end{verbatim}

In the above example, \texttt{buffer} is set to NULL if the function fails, and no longer points to the old memory block. If \texttt{buffer} is your only pointer to the memory block, then you’ve lost access to this memory.

The \texttt{__srealloc()} function reallocates memory from the heap.

You must use \texttt{__sfree()} to deallocate the memory allocated by \texttt{__srealloc()}.

\textbf{Returns:}

A pointer to the start of the reallocated memory, or NULL if there’s insufficient memory available, or if the value of the \texttt{new_size} argument is zero.

\textbf{Classification:}

QNX Neutrino
_srealloc()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

calloc(), free(), realloc(), _scalloc(), _sfree(), _smalloc()
sscanf()  
Scan input from a character string

Synopsis:

```c
#include <stdio.h>

int sscanf( const char* in_string,
            const char* format,
            ... );
```

Arguments:

- `in_string`  
  The string that you want to read from.

- `format`  
  A string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sscanf()` function scans input from the character string `in_string`, under control of the argument `format`. Following the format string is the list of addresses of items to receive values.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is terminated by reaching the end of the input string.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
```
int day, year;
char weekday[20], month[20];

sscanf( "Thursday February 0025 1999",
       "%s %s %d %d",
       weekday, month, &day, &year );
printf( "%s %d, %d is a %s\n",
       month, day, year, weekday );
return EXIT_SUCCESS;
}

produces the following:

February 25, 1999 is a Thursday

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

fscanf(), fwscanf(), scanf(), swscanf(), vfscanf(), vfwscanf(), vscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()
Synopsis:

```
#include <sys/stat.h>

int stat( const char * path,
          struct stat * buf );

int stat64( const char * path,
             struct stat64 * buf );
```

Arguments:

- `path` The path of the file or directory that you want information about.
- `buf` A pointer to a buffer where the function can store the information; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `stat()` and `stat64()` functions obtain information about the file or directory referenced in `path`. This information is placed in the structure located at the address indicated by `buf`.

`stat` structure

Here’s the `stat` structure that’s defined in `<sys/stat.h>`:

```c
struct stat {
  #if _FILE_OFFSET_BITS - 0 == 64
    ino_t st_ino; /* File serial number. */
    off_t st_size; /* File size in bytes. */
  #elif !defined(_FILE_OFFSET_BITS) || _FILE_OFFSET_BITS == 32
    #if defined(__LITTLE_ENDIAN__)  // Big endianness.
      ino_t st_ino; /* File serial number. */
      ino_t st_ino_hi;
    #endif
    off_t st_size;
    off_t st_size_hi;
  #else
    #if defined(__BIG_ENDIAN__)  // Little endianness.
      ino_t st_ino; /* File serial number. */
      ino_t st_ino_hi;
    #endif
    off_t st_size;
    off_t st_size_hi;
  #endif
};
```

September 10, 2007
```
#else
  #error endian not configured for system
#endif
#endif

```

### Access permissions

The access permissions for the file or directory are specified as a combination of bits in the `st_mode` field of a `stat` structure. These bits are defined in `<sys/stat.h>`, and are described below:

<table>
<thead>
<tr>
<th>Owner</th>
<th>Group</th>
<th>Others</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IRUSR</td>
<td>S_IWGRP</td>
<td>S_IWOTH</td>
<td>Read</td>
</tr>
</tbody>
</table>

*continued...*
The following bits define miscellaneous permissions used by other implementations:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IEXEC</td>
<td>S_IXUSR</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>S_IRUSR</td>
</tr>
<tr>
<td>S_IWRITE</td>
<td>S_IWUSR</td>
</tr>
</tbody>
</table>

**st_mode bits**

The following bits are also encoded in the st_mode field:

- **S_ISUID** Set user ID on execution. The process’s effective user ID is set to that of the owner of the file when the file is run as a program. On a regular file, this bit should be cleared on any write.

- **S_ISGID** Set group ID on execution. Set effective group ID on the process to the file’s group when the file is run as a program. On a regular file, this bit should be cleared on any write.
Macros

The following symbolic names for the values of \texttt{st\_mode} are defined for these file types:

\begin{itemize}
  \item \texttt{S\_IFBLK} Block special.
  \item \texttt{S\_IFCHR} Character special.
  \item \texttt{S\_IFDIR} Directory.
  \item \texttt{S\_IFIFO} FIFO special.
  \item \texttt{S\_IFLNK} Symbolic link.
  \item \texttt{S\_IFMT} Type of file.
  \item \texttt{S\_IFNAM} Special named file.
  \item \texttt{S\_IFREG} Regular.
  \item \texttt{S\_IFSOCK} Socket.
\end{itemize}

The following macros test whether a file is of a specified type. The value \texttt{m} supplied to the macros is the value of the \texttt{st\_mode} field of a \texttt{stat} structure. The macros evaluate to a nonzero value if the test is true, and zero if the test is false.

\begin{itemize}
  \item \texttt{S\_ISBLK}(\texttt{m}) Test for block special file.
  \item \texttt{S\_ISCHR}(\texttt{m}) Test for character special file.
  \item \texttt{S\_ISDIR}(\texttt{m}) Test for directory file.
  \item \texttt{S\_ISFIFO}(\texttt{m}) Test for FIFO.
  \item \texttt{S\_ISLNK}(\texttt{m}) Test for symbolic link.
  \item \texttt{S\_ISNAM}(\texttt{m}) Test for special named file.
  \item \texttt{S\_ISREG}(\texttt{m}) Test for regular file.
\end{itemize}
$S_{ISSOCK}(m)$ Test for socket.

These macros test whether a file is of the specified type. The value of the $buf$ argument supplied to the macros is a pointer to a $\text{stat}$ structure. The macro evaluates to a nonzero value if the specified object is implemented as a distinct file type and the specified file type is contained in the $\text{stat}$ structure referenced by the pointer $buf$. Otherwise, the macro evaluates to zero.

$S_{TYPEISMQ}(buf)$
Test for message queue.

$S_{TYPEISSEM}(buf)$
Test for semaphore.

$S_{TYPEISSHM}(buf)$
Test for shared memory object.

These macros manipulate device IDs:

$major(\text{device})$
Extract the major number from a device ID.

$minor(\text{device})$
Extract the minor number from a device ID.

$makedev(\text{node, major, minor})$
Build a device ID from the given numbers. Currently, the $\text{node}$ argument isn’t used and must be zero.

The $st_{rdev}$ member of the $\text{stat}$ structure is a device ID that consists of:

- a major number in the range 0 through 63
- a minor number in the range 0 through 1023.


Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EACCES Search permission is denied for a component of path.

EIO A physical error occurred on the block device.

ELOOP Too many levels of symbolic links or prefixes.

ENAMETOOLONG The argument path exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX. These manifests are defined in the <limits.h> header file.

ENOENT The named file doesn’t exist, or path is an empty string.

ENOSYS The stat() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of path isn’t a directory.

EOVERFLOW The file size in bytes or the number of blocks allocated to the file or the file serial number can’t be represented correctly in the structure pointed to by buf.

Examples:

Determine the size of a file:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
```


```c
int main( void )
{
    struct stat buf;
    if( stat( "file", &buf ) != -1 ) {
        printf( "File size = %d\n", buf.st_size);
    } return EXIT_SUCCESS;
}
```

Determine the amount of free memory:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main () {  
    struct stat buf;
    if ( stat( "/proc", &buf ) == -1) {
        perror ("stat" );
        return EXIT_FAILURE;
    } else {
        printf ("Free memory: %d bytes\n", buf.st_size);
        return EXIT_SUCCESS;
    }
}
```

**Classification:**

`stat()` is POSIX 1003.1; `stat64()` is Large-file support

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`errno`, `fstat()`, `fstat64()`, `lstat()`
statvfs(), statvfs64()

Get filesystem information, given a path

Synopsis:

```c
#include <sys/statvfs.h>

int statvfs( const char *path,  
             struct statvfs *buf );

int statvfs64( const char *path,  
                struct statvfs64 *buf );
```

Arguments:

- `path` The name of a file that resides on the filesystem.
- `buf` A pointer to a buffer where the function can store the information.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `statvfs()` function returns a “generic superblock” describing a filesystem; it can be used to acquire information about mounted filesystems. The `statvfs64()` function is a 64-bit version of `statvfs()`.

The filesystem type is known to the operating system. You don’t need to have read, write, or execute permission for the named file, but all directories listed in the path name leading to the file must be searchable.

The `buf` argument is a pointer to a `statvfs` or `statvfs64` structure that’s filled by the function. It contains at least:

```
unsigned long f_bsize
```

The preferred filesystem blocksize.
unsigned long _f_frsize
The fundamental filesystem blocksize (if supported)

fsblkcnt_t _f_blocks
The total number of blocks on the filesystem, in units of _f_frsize.

fsblkcnt_t _f_bfree
The total number of free blocks.

fsblkcnt_t _f_bavail
The number of free blocks available to a nonsuperuser.

fsfilcnt_t _f_files
The total number of file nodes (inodes).

fsfilcnt_t _f_ffree
The total number of free file nodes.

fsfilcnt_t _f_favail
The number of inodes available to a nonsuperuser.

unsigned long _f_fsid
The filesystem ID (dev for now).

char _f_basetype[16]
The type of the target filesystem, as a null-terminated string.

unsigned long _f_flag
A bitmask of flags; the function can set these flags:
  • ST_RDONLY — read-only filesystem.
  • ST_NOSUID — the filesystem doesn’t support setuid/setgid semantics.

unsigned long _f_namemax
The maximum filename length.
statvfs(), statvfs64()

Returns:

0  Success.
-1  An error occurred (errno is set).

Errors:

EACCES  Search permission is denied on a component of the path prefix.
EFAULT  The path or buf argument points to an illegal address.
EINTR   A signal was caught during execution.
EIO      An I/O error occurred while reading the filesystem.
ELOOP    Too many symbolic links were encountered in translating path.
EMULTIHOP  Components of path require hopping to multiple remote machines and the filesystem type doesn’t allow it.
ENAMETOOLONG  The length of a path component exceeds \{NAME_MAX\} characters, or the length of path exceeds \{PATH_MAX\} characters.
ENOENT   Either a component of the path prefix or the file referred to by path doesn’t exist.
ENOLINK  The path argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR  A component of the path prefix of path isn’t a directory.
EOVERFLOW  One of the values to be returned can’t be represented correctly in the structure pointed to by buf.
**Classification:**

*statvfs()* is POSIX 1003.1 XSI; *statvfs64()* is Large-file support

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

The values returned for *f_files*, *f_ffree*, and *f_favail* might not be valid for NFS-mounted filesystems.

**See also:**

*chmod(), chown(), creat(), dup(), fcntl(), fstatvfs(), fstatvfs64(), link(), mknod(), open(), pipe(), read(), time(), unlink(), utime(), write()*
Synopsis:

```c
#include <stdio.h>

FILE * stderr;
```

Description:

This global variable defines the standard error stream. It’s set to the console by default, but you can redirect it by calling `freopen()`.

`STDERR_FILENO`, which is defined in `<unistd.h>`, defines the file descriptor that corresponds to `stderr`.

Classification:

ANSI, POSIX 1003.1

See also:

`assert()`, `err()`, `errx()`, `getopt()`, `herror()`, `perror()`, `stdin`, `stdout`, `strerror()`, `verr()`, `verrx()`, `vwarn()`, `vwarnx()`, `warn()`, `warnx()`
**Synopsis:**

```c
#include <stdio.h>

FILE * stdin;
```

**Description:**

This global variable defines the standard input stream. It’s set to the console by default, but you can redirect it by calling `freopen()`.

`STDIN_FILENO`, which is defined in `<unistd.h>`, defines the file descriptor that corresponds to `stdin`.

**Classification:**

ANSI, POSIX 1003.1

**See also:**

`fgetchar()`, `getchar()`, `getchar_unlocked()`, `gets()`, `getwchar()`, `scanf()`, `stderr`, `stdout`, `vscanf()`, `vwscanf()`
Synopsis:
```c
#include <stdio.h>

FILE * stdout;
```

Description:
This global variable defines the standard output stream. It’s set to the console by default, but you can redirect it by calling `freopen()`.

`STDOUT_FILENO`, which is defined in `<unistd.h>`, defines the file descriptor that corresponds to `stdout`.

Classification:
ANSI, POSIX 1003.1

See also:
`fputchar()`, `printf()`, `putchar()`, `putchar_unlocked()`, `puts()`, `putwchar()`, `stderr`, `stdin`, `vprintf()`, `vwprintf()`, `wprintf()`


**straddstr()**

*Concatenate one string on to the end of another*

**Synopsis:**

```c
#include <string.h>

int straddstr( const char * str,
               int len,
               char ** pbuf,
               size_t * pmaxbuf );
```

**Arguments:**

- `str` The string that you want to add to the end of another.
- `len` The number of characters from `str` that you want to add. If zero, the function adds all of `str`.
- `pbuf` The address of a pointer to the destination buffer.
- `pmaxbuf` A pointer to the size of the destination buffer.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `straddstr()` function adds `str` to the buffer pointed to by `pbuf`, respecting the maximum length indicated by `pmaxbuf`. The values of `pbuf` and `pmaxlen` are also updated.

**Returns:**

The value of `len` if it’s nonzero; otherwise, the length of `str` (i.e. `strlen(str)`).


Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

strcat(), strcpy(), strncat(), strncpy()
strcasecmp()

Compare two strings, ignoring case

Synopsis:

```
#include <strings.h>

int strcasecmp( const char* str1,
                const char* str2 );
```

Arguments:

str1, str2  The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcasecmp()` function compares two strings, specified by `str1` and `str2`, ignoring the case of the characters.

Returns:

- `< 0`  `s1` is less than `s2`.
- `0`  `s1` is equal to `s2`.
- `> 0`  `s1` is greater than `s2`.

Examples:

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>

void compare( const char* s1, const char* s2 )
{
    int retval;

    retval = strcasecmp( s1, s2 );
    if( retval > 0 ) {
        printf( "%s > %s\n", s1, s2 );
    }
```

3208  C Library — S  September 10, 2007
```c
} else if( retval < 0 ) {
    printf( "%s < %s\n", s1, s2 );
} else {
    printf( "%s == %s\n", s1, s2 );
}

int main( void )
{
    char* str1 = "abcdefg";
    char* str2 = "HIJ";
    char* str3 = "Abc";
    char* str4 = "aBCDEfg";
    compare( str1, str2 );
    compare( str1, str3 );
    compare( str1, str4 );
    compare( str1, str1 );
    compare( str2, str2 );
    compare( str2, str3 );
    compare( str2, str4 );
    compare( str2, str1 );

    return EXIT_SUCCESS;
}

This code produces output that looks like:

abcdefg < HIJ
abcdefg > Abc
abcdefg == aBCDEfg
abcdefg == abcdefg
HIJ == HIJ
HIJ == Abc
HIJ > aBCDEfg
HIJ > abcdefg

Classification:

POSIX 1003.1 XSI

Safety

Cancellation point  No

continued…
### strcasecmp()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`strcmp()`, `strcmpi()`, `strcoll()`, `stricmp()`, `strncasecmp()`, `strnicmp()`, `wcscmp()`, `wcscoll()`, `wcsncmp()`
strcat()

Concatenate two strings

Synopsis:

```c
#include <string.h>

char* strcat( char* dst, const char* src );
```

Arguments:

- `dst`, `src`  The strings that you want to concatenate.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcat()` function appends a copy of the string pointed to by `src` (including the terminating NUL character) to the end of the string pointed to by `dst`. The first character of `src` overwrites the NUL character at the end of `dst`.

Returns:

The same pointer as `dst`.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
    strcpy( buffer, "Hello ");
    strcat( buffer, "world" );
    printf( "%s\n", buffer );
    return EXIT_SUCCESS;
}
```
} produces the output:

Hello world

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

strncat(), strncpy(), strcpy()
strchr()

Find the first occurrence of a character in a string

Synopsis:

```c
#include <string.h>

char* strchr(char* s, int c);
```

Arguments:

- `s`  The string that you want to search.
- `c`  The character that you’re looking for.

Library:

```
libc
```

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `strchr()` function finds the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating NUL character is considered to be part of the string.

Returns:

A pointer to the located character, or NULL if `c` doesn’t occur in the string.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
    char* where;

    strcpy( buffer, "video x-rays" );
```
where = strchr( buffer, 'x' );

if( where == NULL ) {
    printf( "'x' not found\n" );
} else {
    printf( "'x' found: %s\n", where );
}

return EXIT_SUCCESS;

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`, `strtok()`,
`strtok_r()`, `wcschr()`, `wcsstrchr()`, `wcsstr()`, `wcsxstr()`,
Synopsis:

```c
#include <string.h>

int strcmp( const char* s1,
             const char* s2 );
```

Arguments:

s1, s2  The strings that you want to compare.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcmp()` function compares the string pointed to by `s1` to the string pointed to by `s2`.

Returns:

- `< 0`  `s1` is less than `s2`.
- `0`  `s1` is equal to `s2`.
- `> 0`  `s1` is greater than `s2`.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", strcmp( "abcdef", "abcdef" ) );
    printf( "%d\n", strcmp( "abcdef", "abc" ) );
    printf( "%d\n", strcmp( "abc", "abcdef" ) );
    printf( "%d\n", strcmp( "abcdef", "mnopqr" ) );
    printf( "%d\n", strcmp( "mnopqr", "abcdef" ) );
}
```
`strcmp()`

```c
return EXIT_SUCCESS;
}
```

produces the output:

```
0
1
-1
-1
1
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`strcasecmp()`, `strcmpl()`, `strcoll()`, `stricmp()`, `strncompare()`, `strncmp()`, `strnicmp()`, `wcscmp()`, `wcscoll()`, `wcsncmp()`
Synopsis:

```c
#include <string.h>

int strcmpi( const char* s1, const char* s2 );
```

Arguments:

- `s1, s2` The strings that you want to compare.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcmpi()` function compares the string pointed to by `s1` to the string pointed to by `s2`, ignoring case.

All uppercase characters from `s1` and `s2` are mapped to lowercase for the purposes of doing the comparison. The `strcmpi()` function is identical to the `stricmp()` function.

Returns:

- `< 0` `s1` is less than `s2`.
- `0` `s1` is equal to `s2`.
- `> 0` `s1` is greater than `s2`.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", strcmpi( "AbCDEF", "abcdef" ) );
}
```
```c
#include <stdio.h>

int main(void)
{
    int i;

    printf( "%d\n", strcmpi( "abcdef", "ABC" ) );
    printf( "%d\n", strcmpi( "abc", "ABCdef" ) );
    printf( "%d\n", strcmpi( "Abcdef", "mnopqr" ) );
    printf( "%d\n", strcmpi( "Mnopqr", "abcdef" ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0
100
-100
-12
12
```

**Classification:**

- **QNX 4**

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `strcasecmp()`, `strcmp()`, `strcoll()`, `stricmp()`, `strncmp()`, `strncasecmp()`, `strncompare()`, `strnicmp()`, `wcscmp()`, `wcscoll()`, `wcsncmp()`
Synopsis:

```c
#include <string.h>

int strcoll( const char* s1,
              const char* s2 );
```

Arguments:

- `s1, s2` The strings that you want to compare.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcoll()` function compares the strings pointed to by `s1` and `s2`, using the collating sequence selected by the `setlocale()` function.

The `strcoll()` function is equivalent to `strcmp()` when the collating sequence is selected from the "C" locale.

Returns:

- `< 0` `s1` is less than `s2`.
- `0` `s1` is equal to `s2`.
- `> 0` `s1` is greater than `s2`.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char buffer[80] = "world";

int main( void )
{...
```
if( strcoll( buffer, "Hello" ) < 0 ) {
    printf( "Less than\n" );
}

return EXIT_SUCCESS;

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

setlocale(), strcasecmp(), strcmp(), strcmpi(), stricmp(),
strncasecmp(), strncmp(), strnicmp(), wcscmp(), wcsncmp(), wcscoll(), wcscmp(), wcscoll(), wcscmp()
Synopsis:

```
#include <string.h>

char* strcpy( char* dst, const char* src );
```

Arguments:

- `dst` A pointer to where you want to copy the string.
- `src` The string that you want to copy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcpy()` function copies the string pointed to by `src` (including the terminating NUL character) into the array pointed to by `dst`.

Copying of overlapping objects isn’t guaranteed to work properly. See the `memmove()` function for information on copying objects that overlap.

Returns:

The same pointer as `dst`.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char buffer[80];
```
```c
strcpy( buffer, "Hello " );
strcat( buffer, "world" );
printf( "%s\n", buffer );
return EXIT_SUCCESS;
```
produces the output:

Hello world

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memmove()`, `strdup()`, `strncpy()`, `wcscpy()`, `wcsncpy()`, `wmemmove()`
Count the characters at the beginning of a string that aren’t in a given character set

Synopsis:

```c
#include <string.h>

size_t strcspn( const char* str, const char* charset );
```

Arguments:

- `str` The string that you want to search.
- `charset` The set of characters you want to look for.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strcspn()` function finds the length of the initial segment of the string pointed to by `str` that consists entirely of characters not from the string pointed to by `charset`. The terminating NULL character isn’t considered part of `str`.

Returns:

The length of the initial segment.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", strcspn( "abcbcadef", "cba" ) );
    printf( "%d\n", strcspn( "xxxbcadef", "cba" ) );
    printf( "%d\n", strcspn( "123456789", "cba" ) );
    return EXIT_SUCCESS;
}
```
**strcspn()**

produces the output:
0
3
9

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memchr()`, `strchr()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcspbrk()`, `wcsrcr()`, `wcspn()`, `wcsstr()`, `wcstok()`
Synopsis:

```
#include <string.h>

char* strdup( const char* src );
```

Arguments:

`src` The string that you want to copy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strdup()` function creates a duplicate of the string pointed to by `src`, and returns a pointer to the new copy.

The `strdup()` function allocates the memory for the new string by calling `malloc()`; it's up to you to release the memory by calling `free()`.

Returns:

A pointer to a copy of the string, or NULL if an error occurred.

Examples:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    char *dup;

    dup = strdup( "Make a copy" );
    printf( "%s\n", dup );
    free (dup);
}
```
return EXIT_SUCCESS;
}

**Classification:**

POSIX 1003.1 XSI

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`free()`, `malloc()`, `memmove()`, `strcpy()`, `strncpy()`, `wcsncpy()`, `wmemmove()`


Synopsis:

```c
#include <string.h>

char* strerror( int errnum );
```

Arguments:

`errnum` The error number that you want the message for. This function works for any valid `errno` value.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strerror()` function maps the error number contained in `errnum` to an error message.

Returns:

A pointer to the error message.

Don’t modify the string that this function returns.

Examples:

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

int main( void )
{
    FILE *fp;

    fp = fopen( *file.name*, "r" );
    if( fp == NULL ) {
        printf( "Unable to open file: %s\n", ...
```
strerror()
**Synopsis:**

```c
#include <time.h>

size_t strftime( char *s, size_t maxsize, const char *format, const struct tm *timeptr );
```

**Arguments:**

- `s` A pointer to a buffer where the function can store the formatted time.
- `maxsize` The maximum size of the buffer.
- `format` The format that you want to use for the time; see “Formats,” below.
- `timeptr` A pointer to a `tm` structure that contains the time that you want to format.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strftime()` function formats the time in the argument `timeptr` into the array pointed to by the argument `s`, according to the `format` argument.

**Formats**

The `format` string consists of zero or more directives and ordinary characters. A directive consists of a `%` character followed by a character that determines the substitution that’s to take place. All ordinary characters are copied unchanged into the array. No more than `maxsize` characters are placed in the array.
Local timezone information is used as if from a call to `tzset()`.

- `%a` Locale’s abbreviated weekday name.
- `%A` Locale’s full weekday name.
- `%b` Locale’s abbreviated month name.
- `%B` Locale’s full month name.
- `%c` Locale’s appropriate date and time representation.
- `%d` Day of the month as a decimal number (01-31).
- `%D` Date in the format `mm/dd/yy`.
- `%e` Day of the month as a decimal number (1-31); single digits are preceded by a space.
- `%F` The ISO standard date format; equivalent to `%Y-%m-%d`.
- `%g` The last two digits of the week-based year as a decimal number (00-99).
- `%G` The week-based year as a decimal number (e.g. 1998).
- `%h` Locale’s abbreviated month name.
- `%H` Hour (24-hour clock) as a decimal number (00-23).
- `%I` Hour (12-hour clock) as a decimal number (01-12).
- `%j` Day of the year as a decimal number (001-366).
- `%m` Month as a decimal number (01-12).
- `%M` Minute as a decimal number (00-59).
- `%n` Newline character.
- `%p` Locale’s equivalent of either AM or PM.
- `%r` 12-hour clock time (01-12) using the AM/PM notation in the format `HH:MM:SS` (AM | PM).
Some of the above conversion specifiers can be modified with the prefix \texttt{E} or \texttt{O}. If alternative formats don’t exist for the locale, they behave as if the unmodified conversion specifiers were called:
strftime()

%Ec  Alternative date and time representation.
%EC  Alternative name of the the base year (period).
%Ex  Alternative date representation.
%EX  Alternative time representation.
%Ey  Offset from %EC of the alternative year (only) representation.
%Ey  Alternative year representation.
%Od  Day of the month using alternative numeric symbols.
          Leading zeros are added if an alternative symbol for zero exists, otherwise leading spaces are used.
%Od  Day of the month using alternative numeric symbols.
          Leading spaces are used.
%OH  24-hour clock using alternative numeric symbols.
%OI  12-hour clock using alternative numeric symbols.
%Om  Month using alternative numeric symbols.
%OM  Minutes using alternative numeric symbols.
%OS  Seconds using alternative numeric symbols.
%Ou  Alternative week day number representation (Monday=1).
%OU  Alternative week day number representation (Rules correspond with %u).
%OV  Alternative week number representation. (Rules correspond with %v).
%OW  Weekday as a number using alternative numeric symbols (Sunday=0).
%OW  Week number of the year using alternative numeric symbols (Monday is the first day of the week).
%Oy  Year offset from %C using alternative numeric symbols.
Returns:

The number of characters placed into the array, not including the terminating null character, or 0 if the number of characters exceeds maxsize; the string contents are indeterminate. If an error occurs, errno is set.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    time_t time_of_day;
    char buffer[ 80 ];

    time_of_day = time( NULL );
    strftime( buffer, 80, "Today is %A %B %d, %Y",
             localtime( &time_of_day ) );
    printf( "%s\n", buffer );

    return EXIT_SUCCESS;
}
```

This produces the output:

```
Today is Thursday February 25, 1999
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), ctime(), ctime_r(), sprintf(), tm, tzset(), wcsftime()
Synopsis:

#include <string.h>

int stricmp( const char* s1, const char* s2 );

Arguments:

s1, s2 The strings that you want to compare.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `stricmp()` function compares, with case insensitivity, the string pointed to by `s1` to the string pointed to by `s2`. All uppercase characters from `s1` and `s2` are mapped to lowercase for the purposes of doing the comparison.

Returns:

< 0 s1 is less than s2.
0 s1 is equal to s2.
> 0 s1 is greater than s2.

Examples:

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
    printf( "%d\n", stricmp( "AbCDEF", "abcdef" ) );
    printf( "%d\n", stricmp( "abcdef", "ABC" ) );
    printf( "%d\n", stricmp( "abc", "ABCDef" ) );
}
```c
int main(int argc, char **argv)
{
    printf( "%d\n", stricmp( "Abcdef", "mnopqr" ) );
    printf( "%d\n", stricmp( "Mnopqr", "abcdef" ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0
100
-100
-12
12
```

**Classification:**

QNX 4

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`strcasecmp()`, `strcmp()`, `strcmpi()`, `strcoll()`, `strncasecmp()`, `strncmp()`, `strnicmp()`, `wcscmp()`, `wcscoll()`, `wcsncmp()`
**strlen()**

*Compute the length of a string*

**Synopsis:**
```c
#include <string.h>
size_t strlen( const char * s );
```

**Arguments:**
- `s` The string whose length you want to calculate.

**Library:**
```
libc
```
Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**
The `strlen()` function computes the length of the string pointed to by `s`.

**Returns:**
The number of characters that precede the terminating null character.

**Examples:**
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%d\n", strlen( "Howdy" ) );
    printf( "%d\n", strlen( "Hello world\n" ) );
    printf( "%d\n", strlen( "" ) );

    return EXIT_SUCCESS;
}
```

produces the output:
```
5
12
0
```
**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

wcslen()
strlwr()

Convert a string to lowercase

Synopsis:

```c
#include <string.h>

char* strlwr( char* s1 );
```

Arguments:

- `s1` The string that you want to convert to lowercase.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strlwr()` function replaces the string `s1` with lowercase characters, by invoking `tolower()` for each character in the string.

Returns:

The address of the string.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A mixed-case STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strlwr( source ) );
    printf( "%s\n", source );
    return EXIT_SUCCESS;
}
```

produces the output:

A mixed-case STRING
a mixed-case string
a mixed-case string
**Classification:**

Unix

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

strupr(), tolower()
Synopsis:

```c
#include <strings.h>

int strncasecmp( const char* str1,
                 const char* str2,
                 size_t n );
```

Arguments:

- `str1, str2` The strings that you want to compare.
- `n` The maximum number of characters that you want to compare.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strncasecmp()` function compares up to `n` characters in two strings, specified by `str1` and `str2`, ignoring the case of the characters.

Returns:

- `< 0` `str1` is less than `str2`.
- `0` `str1` is equal to `str2`.
- `> 0` `str1` is greater than `str2`.

Examples:

The following code:

```c
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
```

September 10, 2007  C Library — S  3241
void compare( const char *s1, const char *s2 )
{
    int retval;
    retval = strncasecmp( s1, s2, 3 );
    if( retval > 0 ) {
        printf( "%s > %s\n", s1, s2 );
    } else if( retval < 0 ) {
        printf( "%s < %s\n", s1, s2 );
    } else {
        printf( "%s == %s\n", s1, s2 );
    }
}

int main( void )
{
    char *str1 = "abcdefg";
    char *str2 = "HIJ";
    char *str3 = "Abc";
    char *str4 = "aBCDEfg";
    compare( str1, str2 );
    compare( str1, str3 );
    compare( str1, str4 );
    compare( str1, str1 );
    compare( str2, str2 );
    compare( str2, str3 );
    compare( str2, str4 );
    compare( str2, str1 );

    return EXIT_SUCCESS;
}

produces output that looks like:

abcdefg < HIJ
abcdefg == Abc
abcdefg == aBCDEfg
abcdefg == abcdefg
HIJ == HIJ
HIJ > Abc
HIJ > aBCDEfg
HIJ > abcdefg
strncasecmp()

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

strcasecmp(), strcmp(), strcmpi(), strcoll(), stricmp(), strncmp(),
strnicmp(), wcscmp(), wscoll(), wcsncmp()
**Synopsis:**

```c
#include <string.h>

char* strncat( char* dst, const char* src, size_t n );
```

**Arguments:**

- `dst, src` The strings that you want to concatenate.
- `n` The maximum number of characters that you want to add from the `src` string.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strncat()` function appends no more than `n` characters of the string pointed to by `src` to the end of the string pointed to by `dst`. The first character of `src` overwrites the null character at the end of `dst`. This function always adds a terminating null character to the result.

**Returns:**

The same pointer as `dst`.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char buffer[80];

int main( void )
{
    strcpy( buffer, "Hello " );
```
strncat( buffer, "world", 8 );
printf( "%s\n", buffer );
strncat( buffer, "*************", 4 );
printf( "%s\n", buffer );
return EXIT_SUCCESS;
}

produces the output:

Hello world
Hello world****

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

strcat()
**Synopsis:**
```
#include <string.h>

int strncmp( const char* s1,  
    const char* s2,  
    size_t n );
```

**Arguments:**
- `s1`, `s2` The strings that you want to compare.
- `n` The maximum number of characters that you want to compare.

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `strncmp()` function compares up to `n` characters from the strings pointed to by `s1` and `s2`.

**Returns:**
- `< 0` `s1` is less than `s2`.
- `0` `s1` is equal to `s2`.
- `> 0` `s1` is greater than `s2`.

**Examples:**
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main( void )
{
```

3246 C Library — S September 10, 2007
```c
printf( "\n", strncmp( "abcdef", "abcDEF", 10 ) );
printf( "\n", strncmp( "abcdef", "abcDEF", 6 ) );
printf( "\n", strncmp( "abcdef", "abcDEF", 3 ) );
printf( "\n", strncmp( "abcdef", "abcDEF", 0 ) );
return EXIT_SUCCESS;
}
```

produces the output:

```
1
1
0
0
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`strcasestr()`, `strchr()`, `strcspn()`, `strcoll()`, `strcspn()`, `strncasestr()`,
`strchr()`, `strcspn()`, `strcoll()`, `strncasestr()`
**Synopsis:**

```c
#include <string.h>

char* strncpy( char* dst, const char* src, size_t n );
```

**Arguments:**

- `dst` A pointer to where you want to copy the string.
- `src` The string that you want to copy.
- `n` The maximum number of characters that you want to copy.

**Library:**

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strncpy()` function copies no more than `n` characters from the string pointed to by `src` into the array pointed to by `dst`.

Copying of overlapping objects isn’t guaranteed to work properly. See the `memmove()` function if you wish to copy objects that overlap.

If the string pointed to by `src` is shorter than `n` characters, null characters are appended to the copy in the array pointed to by `dst`, until `n` characters in all have been written. If the string pointed to by `src` is longer than `n` characters, then the result isn’t terminated by a null character.
Returns:

The same pointer as dst.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char buffer[15];

    printf( "%s\n", strncpy( buffer, "abcdefg", 10 ) );
    printf( "%s\n", strncpy( buffer, "1234567", 6 ) );
    printf( "%s\n", strncpy( buffer, "abcdefg", 3 ) );
    printf( "%s\n", strncpy( buffer, "*******", 0 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
abcdefg
123456g
abc456g
abc456g
```

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**strncpy()**

See also:

`memmove()`, `strcpy()`, `strdup()`, `wcsncpy()`, `wcscpy()`, `wmemmove()`
**Synopsis:**

```c
#include <string.h>

int strnicmp( const char* s1, const char* s2, size_t len );
```

**Arguments:**

- `s1, s2` The strings that you want to compare.
- `len` The maximum number of characters that you want to compare.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strnicmp()` function compares up to `len` characters from the strings pointed to by `s1` and `s2`, ignoring case.

**Returns:**

- `< 0` `s1` is less than `s2`.
- `0` `s1` is equal to `s2`.
- `> 0` `s1` is greater than `s2`.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
```


```c
printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 10 ) );
printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 6 ) );
printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 3 ) );
printf( "%d\n", strnicmp( "abcdef", "ABCXXX", 0 ) );
return EXIT_SUCCESS;
```

produces the output:

```
-20
-20
0
0
```

**Classification:**

QNX 4

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`strcasecmp()`, `strcmp()`, `strcpy()`, `strcoll()`, `strcspn()`, `strncasecmp()`,
`strncpy()`, `wcsncmp()`, `wscopy()`, `wcsnncmp()`
Strnset() — Fill a string with a given character, to a given length

Synopsis:

```c
#include <string.h>

char * strnset( char * s1, int fill, size_t len );
```

Arguments:

- `s1` The string that you want to fill.
- `fill` The value that you want to fill the string with.
- `len` The number of characters to fill.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strnset()` function fills the string `s1` with the value of the argument `fill`, converted to be a character value. When the value of `len` is greater than the length of the string, the entire string is filled. Otherwise, that number of characters at the start of the string are set to the fill character.

Returns:

The address of the string, `s1`.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A sample STRING" };

int main( void )
```

September 10, 2007
{  
printf( "\n", source );
printf( "%s\n", strnset( source, '=', 100 ) );
printf( "%s\n", strnset( source, '\*', 7 ) );  
return EXIT_SUCCESS;
}

produces the output:

A sample STRING
===============
*******========

Classification:

QNX 4

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

strset()
Synopsis:

```c
#include <string.h>

char* strpbrk(char* str, char* charset);
```

Arguments:

- `str` The string that you want to search.
- `charset` The set of characters you want to look for.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strpbrk()` function locates the first occurrence in the string pointed to by `str` of any character from the string pointed to by `charset`.

Returns:

A pointer to the located character, or NULL if no character from `charset` occurs in `str`.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char* p = "Find all vowels";
    while( p != NULL ) {
        printf( "%s\n", p );
        p = strpbrk( p+1, "aeiouAEIOU" );
    }
    return EXIT_SUCCESS;
}
```
strpbrk() produces the output:
Find all vowels
ind all vowels
all vowels
cwels
eels

Classification:
ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memchr(), strchr(), strcspn(), strchr(), strspn(), strstr(), strtok(),
strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcschr(), wcschspn(),
wcsstr(), wcstok()
Synopsis:

```
#include <string.h>

const char* strrchr(const char* s, int c);
```

Arguments:

- `s` The string that you want to search.
- `c` The character that you’re looking for.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strrchr()` function locates the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null character is considered to be part of the string.

Returns:

A pointer to the located character, or a NULL pointer if the character doesn’t occur in the string.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%s\n", strrchr( "abcdeabcde", 'a' ) );
    if( strrchr( "abcdeabcde", 'x' ) == NULL )
        printf( "NULL\n" );
    return EXIT_SUCCESS;
}
```
strrchr() produces the output:

```
abcde
NULL
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memchr( ), strchr( ), strcspn( ), strpbrk( ), strspn( ), strstr( ), strtok( ),
strtok_r( ), wcschr( ), wcscspn( ), wcspbrk( ), wcsrchr( ), wcsspnp( ),
wcsstr( ), wcstok( )`
strrev()  
Reverse a string

Synopsis:
```c
#include <string.h>

char* strrev( char* s1 );
```

Arguments:
- `s1`  The string that you want to reverse.

Library:
- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `strrev()` function replaces the string `s1` with a string whose characters are in the reverse order.

Returns:
The address of the string, `s1`.

Examples:
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A sample STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strrev( source ) );
    printf( "%s\n", strrev( source ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
GNIRTS elpmas A
A sample STRING
```


**Classification:**

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

© 2007, QNX Software Systems GmbH & Co. KG.
Separate a string into pieces marked by given delimiters

Synopsis:

```c
#include <string.h>

char *strsep( char **stringp, char *delim );
```

Arguments:

- `stringp` The address of a pointer to the string that you want to break into pieces; see below.
- `delim` A set of characters that delimit the pieces in the string.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strsep()` function looks in the null-terminated string pointed to by `stringp` for the first occurrence of any character in `delim` and replaces this with a `\0`, records the location of the next character in `*stringp`, then returns the original value of `*stringp`. If no delimiter characters are found, `strsep()` sets `*stringp` to NULL; if `*stringp` is initially NULL, `strsep()` returns NULL.

Returns:

A pointer to the original value of `*stringp`.

Examples:

Parse strings containing runs of whitespace, making up an argument vector:

```c
char inputstring[100];
char **argv[51], **ap = argv, *p, *val;
```
/* set up inputstring */
for (p = inputstring; p != NULL; ) {
    while ((val = strsep(&p, " \	")) != NULL && *val == '\0');
    *ap++ = val;
}
*ap = 0;

Classification:
Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`strtok()`, `strtok_r()`, `wcstok()`
Synopsis:

```c
#include <string.h>

char* strset( char* s1, int fill );
```

Arguments:

- `s1` The string that you want to fill.
- `fill` The value that you want to fill the string with.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strset()` function fills the string pointed to by `s1` with the character `fill`. The terminating null character in the original string remains unchanged.

Returns:

The address of the string, `s1`.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A sample STRING" };

int main( void )
{
    printf("%s\n", source);
    printf("%s\n", strset( source, '=' ) );
    printf("%s\n", strset( source, '*' ) );
    return EXIT_SUCCESS;
}
```
produces the output:
A sample STRING
***************
***************

Classification:
QNX 4

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

strnset()
strsignal()
Return the description of a signal

Synopsis:
#include <string.h>

char *strsignal( int signo );

Arguments:

signo            The signal number that you want the description of.

Library:

libc
Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The strsignal() function returns a pointer to the language-dependent
string describing a signal.

Returns:

A pointer to the description of the signal, or NULL if signo isn’t a
valid signal number. This array will be overwritten by subsequent
calls to strsignal().

Don’t modify the array returned by this function.

Classification:

Unix

Safety
Cancellation point No

continued...
### strsignal()

See also:

`setlocale()`
strspn()

Count the characters at the beginning of a string that are in a given character set

Synopsis:

```c
#include <string.h>

size_t strspn( const char* str,
                const char* charset );
```

Arguments:

- `str` The string that you want to search.
- `charset` The set of characters you want to look for.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strspn()` function computes the length of the initial segment of the string pointed to by `str` that consists of characters from the string pointed to by `charset`. The terminating null character isn’t considered to be part of `charset`.

Returns:

The length of the segment.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%d\n", strspn( "out to lunch", "aeiou" ) );
    printf( "%d\n", strspn( "out to lunch", "xyz" ) );
    return EXIT_SUCCESS;
}
```

produces the output:
Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strtok()`,
`strtok_r()`, `wcschr()`, `wcscspn()`, `wcschr()`, `wcspbrk()`,
`wcsrchr()`, `wcsspnn()`,
`wcsstr()`, `wcstok()`
**strstr()**

*Find one string inside another*

**Synopsis:**

```c
#include <string.h>

char* strstr(char* str, char* substr);
```

**Arguments:**

- `str`  The string that you want to search.
- `substr`  The string that you’re looking for.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strstr()` function locates the first occurrence in the string pointed to by `str` of the sequence of characters (excluding the terminating null character) in the string pointed to by `substr`.

**Returns:**

A pointer to the located string, or NULL if the string isn’t found.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    printf( "%s\n", strstr("This is an example", "is" ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
is is an example
```
**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcschr()`, `wcsrchr()`, `wcspbrk()`, `wcsstr()`
Synopsis:

```c
#include <stdlib.h>

double strtod( const char *ptr,
    char **endptr );
```

Arguments:

- `ptr` A pointer to the string to parse.
- `endptr` If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strtod()` function converts the string pointed to by `ptr` to `double` representation. The function recognizes a string containing the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits containing an optional decimal point
- an optional `e` or `E`, followed by an optionally signed sequence of digits.

The conversion ends at the first unrecognized character. If `endptr` isn’t NULL, a pointer to the unrecognized character is stored in the object `endptr` points to.
Returns:

The converted value. If the correct value would cause overflow, plus
or minus HUGE_VAL is returned according to the sign, and errno is
set to ERANGE. If the correct value would cause underflow, then zero
is returned, and errno is set to ERANGE.

This function returns zero when the input string can’t be converted. If
an error occurs, errno indicates the error detected.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    double pi;

    pi = strtod( "3.141592653589793", NULL );
    printf( "pi=%17.15f
", pi );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`atof()`, `errno`
strtoimax(), strtoumax()

Convert a string into an integer

Synopsis:

```c
#include <inttypes.h>

intmax_t strtoimax ( const char * nptr, char ** endptr, int base );

uintmax_t strtoumax ( const char * nptr, char ** endptr, int base );
```

Arguments:

- `nptr` A pointer to the string to parse.
- `endptr` If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.
- `base` The base of the number being parsed:
  - If `base` is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
  - If `base` isn’t zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.
Library:

```
libc
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
```

Description:

The `strtoimax()` and `strtoumax()` functions are the same as the `strtol()`, `strtoll()`, `stroul()`, and `strtoull()` functions except that they return objects of type `intmax_t` and `uintmax_t`.

Returns:

The converted value.

If the correct value causes an overflow, `INTMAX_MAX` or `INTMAX_MIN` is returned according to the sign and `errno` is set to ERANGE. If `base` is out of range, zero is returned and `errno` is set to EINVAL.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`strtol()`, `stroul()`
Synopsis:

```c
#include <string.h>

char* strtok( char* s1,
              const char* s2 );
```

Arguments:

- `s1`  NULL, or the string that you want to break into tokens; see below.
- `s2`  A set of the characters that separate the tokens.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `strtok()` function breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a character from the string pointed to by `s2`.

The first call to `strtok()` returns a pointer to the first token in the string pointed to by `s1`. Subsequent calls to `strtok()` must pass a NULL pointer as the first argument, in order to get the next token in the string. The set of delimiters used in each of these calls to `strtok()` can be different from one call to the next.

The first call in the sequence searches `s1` for the first character that isn’t contained in the current delimiter string `s2`. If no such character is found, then there are no tokens in `s1`, and `strtok()` returns a NULL pointer. If such a character is found, it’s the start of the first token.

The `strtok()` function then searches from there for a character that’s contained in the current delimiter string. If no such character is found, the current token extends to the end of the string pointed to by `s1`. If such a character is found, it’s overwritten by a null character, which
terminates the current token. The `strtok()` function saves a pointer to the following character, from which the next search for a token will start when the first argument is a NULL pointer.

You might want to keep a copy of the original string because `strtok()` is likely to modify it.

**Returns:**

A pointer to the token found, or NULL if no token was found.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( void )
{
    char* p;
    char* buffer;
    char* delims = { " .," };

    buffer = strdup( "Find words, all of them." );
    printf( "%s\n", buffer );
    p = strtok( buffer, delims );
    while( p != NULL ) {
        printf( "word: %s\n", p );
        p = strtok( NULL, delims );
    }
    printf( "%s\n", buffer );
    return EXIT_SUCCESS;
}
```

produces the output:

```
Find words, all of them.
word: Find
word: words
word: all
word: of
word: them
Find
```
Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Yes/No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strset()`, `strspn()`,
`strstr()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcschr()`, `wcsrchr()`,
`wcspbrk()`, `wcsstr()`, `wcsspn()`, `wcstok()`
strtok_r()

Break a string into tokens (reentrant)

Synopsis:

```
#include <string.h>

char* strtok_r( char* s,
               const char* sep,
               char** lasts );
```

Arguments:

- **s1**: NULL, or the string that you want to break into tokens; see below.
- **s2**: A set of the characters that separate the tokens.
- **lasts**: The address of a pointer to a character, which the function can use to store information necessary for it to continue scanning the same string.

Library:

```
libc
```

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The function **strtok_r()** breaks the string **s** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **sep**.

In the first call to **strtok_r()**, **s** must point to a null-terminated string, **sep** points to a null-terminated string of separator characters, and **lasts** is ignored. The **strtok_r()** function returns a pointer to the first character of the first token, writes a NULL character into **s** immediately following the returned token, and updates **lasts**.

In subsequent calls, **s** must be a NULL pointer and **lasts** must be unchanged from the previous call so that subsequent calls will move through the string **s**, returning successive tokens until no tokens...
remain. The separator string \textit{sep} may be different from call to call. When no tokens remain in \textit{s}, a NULL pointer is returned.

**Returns:**

A pointer to the token found, or NULL if no token was found.

**Classification:**

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

strtol(), strtoll()

Convert a string into a long integer

Synopsis:

```
#include <stdlib.h>

long int strtol( const char * ptr,
                 char ** endptr,
                 int base );

long long strtoll( const char * ptr,
                   char ** endptr,
                   int base );
```

Arguments:

- `ptr` A pointer to the string to parse.
- `endptr` If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.
- `base` The base of the number being parsed:
  - If `base` is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are `0x` or `0X` the digits are treated as hexadecimal. If the first character is `0`, the digits are treated as octal. Otherwise, the digits are treated as decimal.
  - If `base` isn’t zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than `base` are permitted. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
**Description:**

The `strtol()` function converts the string pointed to by `ptr` to an object of type `long int`; `strtoll()` converts the string pointed to by `ptr` to an object of type `long long`.

These functions recognize strings that contain the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits and letters.

The conversion ends at the first unrecognized character. If `endptr` isn’t NULL, a pointer to the unrecognized character is stored in the object `endptr` points to.

**Returns:**

The converted value.

If the correct value causes an overflow, `LONG_MAX` | `LONGLONG_MAX` or `LONG_MIN` | `LONGLONG_MIN` is returned according to the sign, and `errno` is set to `ERANGE`. If `base` is out of range, zero is returned and `errno` is set to `EDOM`.

**Examples:**

```c
#include <stdlib.h>

int main( void )
{
   long int v;

   v = strtol( "12345678", NULL, 10 );
   return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1
**strtol(), strtoll()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`atoi(), atol(), errno, itoa(), ltoa(), sscanf(), strtol(), ultoa(), ultoa()`


**Synopsis:**

```c
#include <stdlib.h>

unsigned long int strtoul( const char * ptr,
    char ** endptr,
    int base );

unsigned long long strtoull( const char * ptr,
    char ** endptr,
    int base );
```

**Arguments:**

- `ptr` A pointer to the string to parse.
- `endptr` If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.
- `base` The base of the number being parsed:
  - If `base` is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
  - If `base` isn’t zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:
The `strtoul()` function converts the string pointed to by `ptr` to an unsigned long; `strtoull()` converts the string pointed to by `ptr` to an unsigned long long.

These functions recognize strings that contain the following:

- optional white space
- a sequence of digits and letters.

The conversion ends at the first unrecognized character. A pointer to that character is stored in the object `endptr` points to, if `endptr` isn’t NULL.

Returns:
The converted value.

If the correct value causes an overflow, `ULONG_MAX | ULONGLONG_MAX` is returned and `errno` is set to `ERANGE`. If `base` is out of range, zero is returned and `errno` is set to `EDOM`.

Examples:
```c
#include <stdlib.h>

int main( void )
{
    unsigned long int v;

    v = strtoul( "12345678", NULL, 10 );
    return EXIT_SUCCESS;
}
```

Classification:
ANSI, POSIX 1003.1
strtol(), strtoull()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

atoi(), atol(), errno, itoa(), ltoa(), sscanf(), strtol(), ultoa(), utoa()
**Synopsis:**

```c
#include <string.h>

char* strupr( char* s1 );
```

**Arguments:**

- `s1` The string that you want to convert to uppercase.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strupr()` function replaces the string `s1` with uppercase characters, by invoking `toupper()` for each character in the string.

**Returns:**

The address of the string.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A mixed-case STRING" };

int main( void )
{
    printf( "%s\n", source );
    printf( "%s\n", strupr( source ) );
    printf( "%s\n", source );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A mixed-case STRING
A MIXED-CASE STRING
A MIXED-CASE STRING
```
Classification:

Unix

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

strlwr(), toupper()
**strxfrm()**

Transform one string into another, to a given length

**Synopsis:**

```c
#include <string.h>

size_t strxfrm( char* dst, const char* src, size_t n );
```

**Arguments:**

- **dst**
  - The string that you want to transform.
- **src**
  - The string that you want to place in *dst*.
- **n**
  - The maximum number of characters to transform.

**Library:**

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strxfrm()` function transforms, for no more than *n* characters, the string pointed to by *src* to the buffer pointed to by *dst*. The transformation uses the collating sequence selected by `setlocale()` so that two transformed strings compare identically (using the `strncmp()` function) to a comparison of the original two strings using `strcoll()`.

If the collating sequence is selected from the "C" locale, `strxfrm()` is equivalent to `strncpy()`, except that `strxfrm()` doesn’t pad the *dst* argument with null characters when the argument *src* is shorter than *n* characters.

**Returns:**

The length of the transformed string. If this length is more than *n*, the contents of the array pointed to by *dst* are indeterminate.
If an error occurs, `strxfrm()` sets `errno` and returns 0. Since the function could also return zero on success, the only way to tell that an error has occurred is to set `errno` to 0 before calling `strxfrm()` and check `errno` afterward.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

char src[] = { "A sample STRING"};
char dst[20];

int main( void )
{
    size_t len;
    setlocale( LC_ALL, "C" );
    printf( "%s\n", src );
    len = strxfrm( dst, src, 20 );
    printf( "%s (%u)\n", dst, len );
    return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
A sample STRING (15)
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
### strxfrm()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`setlocale()`, `strcoll()`, `wcsxfrm()`
**swab()**

**Endian-swap a given number of bytes**

**Synopsis:**

```c
#include <unistd.h>

void swab( const void * src,  
           void * dest,  
           ssize_t nbytes );
```

**Arguments:**

- `src` A pointer to the buffer that you want to copy the bytes from.
- `dest` A pointer to the buffer where you want the function to copy the bytes.
- `nbytes` The number of bytes that you want to copy and swap.

**Library:**

`libc`

Use the `-l c` option to *qcc* to link against this library. This library is usually included automatically.

**Description:**

The `swab()` function copies `nbytes` bytes, pointed to by `src`, to the object pointed to by `dest`, exchanging adjacent bytes. The `nbytes` argument should be even.

<table>
<thead>
<tr>
<th>If <code>nbytes</code> is:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd</td>
<td><code>nbytes</code>-1 bytes are copied and exchanged. The disposition of the last byte is unspecified.</td>
</tr>
<tr>
<td>Negative</td>
<td><code>swab()</code> does nothing.</td>
</tr>
</tbody>
</table>

If copying takes place between objects that overlap, the behavior is undefined.
**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`
swprintf()

Print formatted wide-character output into a string

Synopsis:

```c
#include <wchar.h>

int swprintf( wchar_t * ws,
             size_t n,
             const wchar_t * format,
             ...);
```

Arguments:

- `ws` A pointer to the buffer where you want the function to store the formatted string.
- `n` The maximum number of wide characters to store in the buffer, including a terminating null character.
- `format` A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `swprintf()` function is similar to `fwprintf()` except that `swprintf()` places the generated output into the wide-character array pointed to by `buf`, instead of writing it to a file. A null character is placed at the end of the generated character string.

The `swprintf()` function is the wide-character version of `sprintf()`.
swprintf()

Returns:

The number of wide characters written, excluding the terminating NUL, or a negative number if an error occurred (errno is set).

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fprintf(), fwprintf(), printf(), snprintf(), sprintf(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
Synopsis:

```c
#include <wchar.h>

int swscanf( const wchar_t * ws,
             const wchar_t * format,
             ... );
```

Arguments:

- `ws` The wide-character string that you want to read from.
- `format` A wide-character string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

Library:

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `swscanf()` function scans input from the wide-character string `ws`, under control of the argument `format`. Following the format string is the list of addresses of items to receive values.

The `swscanf()` function is the wide-character version of `sscanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is terminated by reaching the end of the input string.
swscanf()

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

errno, fscanf(), fwscanf(), scanf(), sscanf(), vfscanf(), vfwscanf(), vscanf(), vsscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()
symlink()

Create a symbolic link to a path

Synopsis:

```
#include <unistd.h>

int symlink( const char* pname, const char* slink );
```

Arguments:

- `pname` The path that you want to link to.
- `slink` The symbolic link that you want to create.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `symlink()` function creates a symbolic link named `slink` that contains the pathname specified by `pname` (`slink` is the name of the symbolic link created, `pname` is the pathname contained in the symbolic link).

File access checking isn’t performed on the file named by `pname`, and the file need not exist.

If the `symlink()` function is unsuccessful, any file named by `slink` is unaffected.

Returns:

- `0` Success.
- `-1` An error occurred (`errno` is set).
Errors:

EACCES  A component of the slink path prefix denies search permission, or write permission is denied in the parent directory of the symbolic link to be created.

EEXIST  A file named by slink already exists.

ELOOP  A loop exists in symbolic links encountered during resolution of the slink argument, and it resolves to more than SYMLOOP_MAX levels.

ENAMETOOLONG  A component of the path specified by slink exceeds NAME_MAX bytes, or the length of the entire pathname exceeded PATH_MAX characters.

ENOSPC  The new symbolic link can’t be created because there’s no space left on the filesystem that will contain the symbolic link.

ENOSYS  The symlink() function isn’t implemented for the filesystem specified in slink.

ENOTDIR  A component of the path prefix of slink isn’t a directory.

ERofs  The file slink would reside on a read-only filesystem.

Examples:

/*
 * create a symbolic link to "/usr/nto/include"
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    if( symlink( "/usr/nto/include", "slink" ) == -1) {
        perror( "slink -> /usr/nto/include" );
        exit( EXIT_FAILURE );
    }
}
exit( EXIT_SUCCESS );
}

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, link(), lstat(), pathmgr_symlink(), pathmgr_unlink(),
readlink(), unlink()
**sync()**

Synchronize filesystem updates

**Synopsis:**

```
#include <unistd.h>

void sync( void );
```

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sync()` function queues all the modified block buffers for writing, and returns; it doesn’t wait for the actual I/O to take place. Use this function — or `fsync()` for a single file — to ensure consistency of the entire on-disk filesystem with the contents of the in-memory buffer cache.

**Classification:**

POSIX 1003.1 XSI

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`fdatasync()`, `fsync()`
SyncCondvarSignal(),
SyncCondvarSignal_r()

Wake up any threads that are blocked on a synchronization object

Synopsis:

```c
#include <sys/neutrino.h>

int SyncCondvarSignal( sync_t* sync,
                        int broadcast);

int SyncCondvarSignal_r( sync_t* sync,
                         int broadcast);
```

Arguments:

- **sync**  
  A pointer to a `sync_t` for the synchronization object. You must have initialized this argument by calling `SyncTypeCreate()` or statically initialized it with the manifest PTHREAD_COND_INITIALIZER.

- **broadcast**  
  Zero if you want to make ready to run the thread with the highest priority that's been waiting the longest, or nonzero if you want to make all waiting threads ready.

Library:

- **libc**

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncCondvarSignal()` and `SyncCondvarSignal_r()` kernel calls wake up one or all threads that are blocked on the synchronization object `sync`.

These functions are similar, except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling
pthread_cond_broadcast() or pthread_cond_signal().

In all cases, each awakened thread attempts to reacquire the
controlling mutex passed in SyncCondvarWait() before control is
returned to the thread. If the mutex is already locked when the kernel
attempts to lock it, the thread becomes blocked on the mutex until it’s
unlocked.

Blocking states
These calls don’t block.

Returns:
The only difference between these functions is the way they indicate
errors:

SyncCondvarSignal()
If an error occurs, the function returns -1 and sets errno. Any
other value returned indicates success.

SyncCondvarSignal_r()
EOK is returned on success. This function does NOT set errno.
If an error occurs, any value in the Errors section may be
returned.

Errors:
EFAULT A fault occurred when the kernel tried to access sync.
EINVAL The synchronization ID specified in sync doesn’t exist.

Classification:
QNX Neutrino
SyncCondvarSignal(),
SyncCondvarSignal_r()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_broadcast(), pthread_cond_signal(),
pthread_cond_wait(), SyncCondvarWait()
Block a thread on a synchronization object

Synopsis:

```c
#include <sys/neutrino.h>

int SyncCondvarWait( sync_t * sync,
                      sync_t * mutex );

int SyncCondvarWait_r( sync_t * sync,
                        sync_t * mutex );
```

Arguments:

- `sync` A pointer to a `sync_t` for the synchronization object. You must have initialized this argument by calling `SyncTypeCreate()` or statically initialized it with the manifest `PTHREAD_COND_INITIALIZER`.

- `mutex` The mutex that's associated with the condition variable. You must lock this mutex by calling `SyncMutexLock()` (or the POSIX `pthread_mutex_lock() cover routine). The kernel releases the mutex lock in the kernel when it blocks the thread on `sync`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncCondvarWait()` and `SyncCondvarWait_r()` kernel calls block the calling thread on the synchronization object, `sync`. If more than one thread is blocked on the object, they're queued in priority order.

These functions are similar, except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling `pthread_cond_timedwait()` or `pthread_cond_wait()`.

The blocked thread can be unblocked by any one of the following conditions:

**Condition variable signalled**

The condition variable was signaled by a call to `SyncCondvarSignal()`, that determined that this thread should be awakened.

Before returning from `SyncCondvarWait()`, `mutex` is reacquired. If `mutex` is locked, the thread enters into the STATE_MUTEX state waiting for `mutex` to be unlocked. At this point it’s as though you had called `SyncMutexLock(mutex)`.

**Timeout**

The wait was terminated by a timeout initiated by a previous call to `TimerTimeout()`.

Before returning from `SyncCondvarWait()`, `mutex` is reacquired. If `mutex` is locked, the thread enters into the STATE_MUTEX state waiting for `mutex` to be unlocked. At this point it’s as though you had called `SyncMutexLock(mutex)`.

**POSIX signal**

The wait was terminated by an unmasked signal initiated by a call to `SignalKill()`. If a signal handler has been set up, the signal handler runs with `mutex` unlocked. On return from the signal handler, `mutex` is reacquired. If `mutex` is locked, the thread enters into the STATE_MUTEX state waiting for `mutex` to be unlocked. At this point, it’s as though you had called `SyncMutexLock(mutex)`.

**Thread cancellation**

The wait was terminated by a thread cancellation initiated by a call to `ThreadCancel()`. Before
calling the cancellation handler, \textit{mutex} is reacquired. If \textit{mutex} is locked, the thread enters into the STATE\_MUTEX state waiting for \textit{mutex} to be unlocked. At this point, it’s as though you had called \textit{SyncMutexLock}(\textit{mutex}).

In all cases, \textit{mutex} is reacquired before the call returns. If the thread enters the STATE\_MUTEX state, the rules governing \textit{SyncMutexLock()} are in effect.

Condition variables are used to block a thread until a certain condition is satisfied. Spurious wakeups may occur due to timeouts, signals, and broadcast condition variable signals. Therefore, you should always reevaluate the condition, even on a successful return. The easiest way to do this is with a while loop. For example:

\begin{verbatim}
SyncMutexLock(&mutex);
while(some_condition) {
    SyncCondvarWait(&condvar, &mutex);
}
SyncMutexUnlock(&mutex);
\end{verbatim}

**Blocking states**

**STATE\_CONDVAR**

The calling thread blocks waiting for the condition variable to be signaled.

**STATE\_MUTEX**

The thread was unblocked from the STATE\_CONDVAR state and while trying to reacquire the controlling mutex, found the mutex was locked by another thread.

**Returns:**

The only difference between these functions is the way they indicate errors:
SyncCondvarWait()

If an error occurs, the function returns -1 and sets errno. Any other value returned indicates success.

SyncCondvarWait_r()

Returns EOK on success. This function does NOT set errno. If an error occurs, the function returns any value in the Errors section.

Errors:

- **EAGAIN**: On the first use of a statically initialized sync, all kernel synchronization objects were in use.
- **EFAULT**: A fault occurred when the kernel tried to access sync or mutex.
- **EINVAL**: The synchronization ID specified in sync doesn’t exist.
- **ETIMEDOUT**: A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

pthread_cond_broadcast(), pthread_cond_signal(),
pthread_cond_timedwait(), pthread_cond_wait(),
pthread_mutex_lock(), SignalKill(), SyncCondvarSignal(),
SyncMutexLock(), SyncTypeCreate(), ThreadCancel(),
TimerTimeout()
Perform an operation on a synchronization object

Synopsis:

```c
#include <sys/neutrino.h>

int SyncCtl( int cmd,  
             sync_t * sync,  
             void * data );

int SyncCtl_r( int cmd,  
               sync_t * sync,  
               void * data );
```

Arguments:

- `cmd` The operation type; one of:
  - `_NTO_SCTL_GETPRIOCEILING` — get the ceiling priority of the mutex pointed to by `sync` and put it in the variable pointed to by `data`.
  - `_NTO_SCTL_SETPRIOCEILING` — return the original ceiling priority. Set the ceiling priority of the mutex pointed to by `sync` to the value pointed to by `data`.
  - `_NTO_SCTL_SETEVENT` — attach an event, pointed to by `data`, to the mutex pointed to by `sync`.

You should use `SyncMutexEvent()` to do this.

- `sync` A pointer to the synchronization object that you want to manipulate.

- `data` A pointer to data associated with the command, or a place where the function can store the requested information, depending on the operation.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `SyncCtl()` and `SyncCtl_r()` kernel calls let you:

- set or get a ceiling priority for a mutex
  
  or

- attach an event to a mutex so you’ll be notified when the mutex changes to the DEAD state.

These functions are similar, except for the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `pthread_mutex_getprioceiling()` or `pthread_mutex_setprioceiling()`.

Returns:

The only difference between these functions is the way they indicate errors:

- `SyncCtl()` If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

- `SyncCtl_r()` Returns EOK on success. This function does NOT set `errno`. If an error occurs, the function returns any value listed in the Errors section.

Errors:

- EAGAIN All kernel synchronization event objects are in use.
-EFAULT A fault occurred when the kernel tried to access `sync` or `data`.
-EINVAL The synchronization object pointed to by `sync` doesn’t exist, or the ceiling priority value pointed to by `data` is out of range.
-ENOSYS The `SyncCtl()` and `SyncCtl_r()` functions aren’t currently supported.
SyncCtl(), SyncCtl_r()

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_getprioceiling(), pthread_mutex_setprioceiling(), SyncCondvarSignal(), SyncCondvarWait(), SyncDestroy(), SyncMutexEvent(), SyncMutexLock(), SyncMutexRevive(), SyncMutexUnlock(), SyncTypeCreate()
SyncDestroy(), SyncDestroy_r() © 2007, QNX Software Systems GmbH & Co. KG.

Destroy a synchronization object

Synopsis:

```
#include <sys/neutrino.h>

int SyncDestroy( sync_t* sync );
int SyncDestroy_r ( sync_t* sync );
```

Arguments:

- `sync` The synchronization object that you want to destroy.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncDestroy()` and `SyncDestroy_r()` kernel calls destroy a synchronization object previously allocated by a call to `SyncTypeCreate()`. If the object is a locked mutex, or a condition variable with waiting threads, the call fails. Any attempt to use `sync` after it is destroyed fails.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Don’t call `SyncDestroy()` directly; instead, use the POSIX synchronization objects (see `pthread_cond_destroy()`, `pthread_mutex_destroy()`, `pthread_rwlock_destroy()`, and `sem_destroy()`).

Blocking states

These calls don’t block.
SyncDestroy(),
SyncDestroy_r()

Returns:

The only difference between these functions is the way they indicate errors:

SyncDestroy()

If an error occurs, the function returns -1 and sets errno. Any other value returned indicates success.

SyncDestroy_r()

Returns EOK on success. This function does NOT set errno. If an error occurs, the function can return any value listed in the Errors section.

Errors:

EBUSY     The synchronization object is locked by a thread.
EFAULT    A fault occurred when the kernel tried to access sync.
EINVAL    The synchronization ID specified in sync doesn’t exist.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_destroy(), pthread_mutex_destroy(),
pthread_rwlock_destroy(), sem_destroy(), SyncTypeCreate()
SyncMutexEvent(), SyncMutexEvent_r() © 2007, QNX Software Systems GmbH & Co. KG.

Attach an event to a mutex

Synopsis:

```c
#include <sys/neutrino.h>

int SyncMutexEvent( sync_t * sync,
                    struct sigevent * event );

int SyncMutexEvent_r( sync_t * sync,
                      struct sigevent * event );
```

Arguments:

- `sync` A pointer to the synchronization object for the mutex that you want to attach an event to.
- `event` A pointer to the `sigevent` structure that describes the event that you want to attach.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncMutexEvent()` is a kernel call that attaches a specified `event` to a mutex pointed to by `sync`. You use `SyncMutexRevive()` to revive a DEAD mutex. Normally, a mutex will be placed in the DEAD state when the memory that was used to lock the mutex gets unmapped. One of the ways this may happen is when a process dies while holding the mutex in a shared memory.

`SyncMutexEvent()` and `SyncMutexEvent_r()` are similar, except for the way they indicate errors. See the Returns section for details.
Returns:

The only difference between these functions is the way they indicate errors:

*SyncMutexEvent()*

If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

*SyncMutexEvent_r()*

Returns EOK on success. This function does **NOT** set `errno`. If an error occurs, the function returns any value listed in the Errors section.

Errors:

- EAGAIN: All kernel synchronization event objects are in use.
- EFAULT: A fault occurred when the kernel tried to access `sync`.
- EINVAL: The synchronization object pointed to by `sync` doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

*sigevent*, *SyncCondvarSignal()*, *SyncCondvarWait()*,
*SyncDestroy()*, *SyncMutexLock()*, *SyncMutexRevive()*,
*SyncMutexUnlock()*
SyncMutexLock(),
SyncMutexLock_r()

Lock a mutex synchronization object

Synopsis:

#include <sys/neutrino.h>

int SyncMutexLock( sync_t * sync );

int SyncMutexLock_r( sync_t * sync );

Arguments:

sync A pointer to the synchronization object for the mutex that you want to lock.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The SyncMutexLock() and SyncMutexLock_r() kernel calls try to lock the mutex synchronization object sync. If the mutex isn’t currently locked, the call returns immediately with the object locked. The mutex is considered unlocked if the owner field of sync is zero. Otherwise, the owner field of sync is treated as the thread ID of the current owner of the mutex.

These functions are similar, except for the way they indicate errors. See the Returns section for details.

The POSIX functions pthread_mutex_lock(), and pthread_mutex_unlock(), are faster, since they can potentially avoid a kernel call.

If the mutex is already locked, the calling thread blocks on sync until it’s unlocked by the owner. If more than one thread is blocked on sync they’re queued in priority order.
If the priority of the blocking thread is higher than the thread that owns the mutex, the owner’s priority is boosted to match that of the caller. In other words, the owner inherits the caller’s priority if it’s higher. If the owner’s priority is boosted, it returns to its previous value before any boosts when the mutex is unlocked. Note that the owner may be boosted more than once as higher priority threads block on `sync`.

If a thread is boosted via this mechanism and subsequently changes its own priority, that priority takes immediate effect and also becomes the value it’s returned to after it releases the mutex.

Waiting for a mutex isn’t a cancellation point. If a signal is delivered to the thread while waiting for the mutex, the signal handler runs and, upon return from the handler, the thread resumes waiting for the mutex as if it wasn’t interrupted.

---

Avoid timeouts on mutexes. Mutexes should be locked for brief periods of time, eliminating the need for a timeout.

The `sync` argument must have been initialized by a call to `SyncTypeCreate()` or have been statically initialized by the manifest `PTHREAD_MUTEX_INITIALIZER`.

**Blocking states**

`STATE_MUTEX`

The calling thread blocks waiting for the synchronization object to be unlocked.

**Returns:**

The only difference between these functions is the way they indicate errors:

`SyncMutexLock()`

If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.
SyncMutexLock_r()

Returns EOK on success. This function does NOT set errno. If an error occurs, this function returns any value listed in the Errors section.

Errors:

- EAGAIN: On the first use of a statically initialized sync, all kernel synchronization objects were in use.
- EFAULT: A fault occurred when the kernel tried to access the buffers you provided.
- EINVAL: The synchronization ID specified in sync doesn’t exist.
- ETIMEDOUT: A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_lock(), pthread_mutex_unlock(), SyncTypeCreate(), SyncDestroy(), SyncMutexUnlock()
Revive a mutex that’s in the DEAD state

Synopsis:

```c
#include <sys/neutrino.h>

int SyncMutexRevive( sync_t * sync );

int SyncMutexRevive_r( sync_t * sync );
```

Arguments:

- `sync` A pointer to the synchronization object for the mutex that you want to revive.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncMutexRevive()` and `SyncMutexRevive_r()` kernel calls revive a mutex, pointed to by `sync`, that’s in the DEAD state. The mutex will be put into the LOCK state and will be owned by the calling thread. The mutex counter is set to one (for recursive mutexes).

These functions are similar, except for the way they indicate errors. See the Returns section for details.

See `SyncMutexEvent()` for information on how to get notified when a mutex enters the DEAD state.

Returns:

The only difference between these functions is the way they indicate errors:

- `SyncMutexRevive()`

  If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.
SyncMutexRevive_r()

Returns EOK on success. This function does NOT set errno. If an error occurs, the function returns any value listed in the Errors section.

Errors:

EFAULT A fault occurred when the kernel tried to access the buffers you provided.

EINVAL The synchronization object pointed to by sync doesn’t exist or wasn’t in the DEAD state.

ETIMEDOUT A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_mutex_lock(), pthread_mutex_unlock(), SyncTypeCreate(), SyncDestroy(), SyncMutexEvent(), SyncMutexLock(), SyncMutexUnlock()
Unlock a mutex synchronization object

Synopsis:

```c
#include <sys/neutrino.h>

int SyncMutexUnlock( sync_t * sync );
int SyncMutexUnlock_r( sync_t * sync );
```

Arguments:

- `sync` A pointer to the synchronization object for the mutex that you want to unlock.

Library:

- `libc`

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `SyncMutexUnlock()` and `SyncMutexUnlock_r()` kernel calls unlock the mutex passed as `sync`. If there are threads blocked on the mutex, the `owner` member of `sync` is set to the thread ID of the thread with the highest priority that has been waiting the longest and it’s made ready to run. If no threads are waiting, it’s set to zero.

These functions are similar, except for the way they indicate errors. See the Returns section for details.

If the calling thread had its priority boosted while it owned the mutex, it returns to its normal priority.

The POSIX functions `pthread_mutex_lock()`, and `pthread_mutex_unlock()`, are faster, since they can potentially avoid a kernel call.


Blocking states

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

**SyncMutexUnlock()**

If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

**SyncMutexUnlock_r()**

Returns EOK on success. This function does **NOT** set `errno`. If an error occurs, the function returns any value listed in the Errors section.

**Errors:**

**EFAULT**  
A fault occurred when the kernel tried to access the buffers provided.

**EINVAL**  
The synchronization ID specified in `sync` doesn’t exist. The calling thread doesn’t own the mutex.

**Classification:**

QNX Neutrino

---

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

pthread_mutex_lock(), pthread_mutex_unlock(), SyncTypeCreate(),
SyncMutexLock()
Synopsis:

#include <sys/neutrino.h>

int SyncSemPost( sync_t* sync );
int SyncSemPost_r( sync_t* sync );

Arguments:

sync  A pointer to the synchronization object for the semaphore
that you want to increment.

Library:

libc

Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The SyncSemPost() and SyncSemPost_r() kernel calls increment the
semaphore referenced by the sync argument. If any threads are
blocked on the semaphore, the one waiting the longest is unblocked
and allowed to run.

These functions are identical, except for the way they indicate errors.
See the Returns section for details.

You should use the POSIX sem_post() function instead of calling
SyncSemPost() directly.

Returns:

The only difference between these functions is the way they indicate
errors:

SyncSemPost()

If an error occurs, the function returns -1 and sets errno. Any
other value returned indicates success.
**SyncSemPost(), SyncSemPost_r()**

GmbH & Co. KG.

`SyncSemPost_r()`

Returns EOK on success. This function does **NOT** set `errno`. If an error occurs, the function returns one of the values listed in the Errors section.

**Errors:**

- **EAGAIN** Not enough memory for the kernel to create the internal `sync` object.
- **EFAULT** Invalid pointer.
- **EINTR** A signal interrupted this function.
- **EINVAL** The `sync` argument doesn’t refer to a valid semaphore.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`sem_destroy()`, `sem_init()`, `sem_post()`, `sem_trywait()`, `sem_wait()`, `SyncDestroy()`, `SyncSemWait()`, `SyncTypeCreate()`
Synopsis:

```c
#include <sys/neutrino.h>

int SyncSemWait( sync_t* sync,
    int try );

int SyncSemWait_r( sync_t* sync,
    int try );
```

Arguments:

- `sync` A pointer to the synchronization object for the semaphore that you want to wait on.
- `try` Nonzero if you want a conditional wait.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SyncSemWait()` and `SyncSemWait_r()` kernel calls decrement the semaphore referred to by the `sync` argument. If the semaphore value isn’t greater than zero and `try` is zero, then the calling process blocks until it can decrement the counter or the call is interrupted by signal.

These functions are identical, except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `sem_timedwait()`, `sem_trywait()` or `sem_wait()`.

If `try` is nonzero, the function acts as a conditional wait. If the call would block, the semaphore is unmodified, and the call returns with an error.
SyncSemWait(), SyncSemWait_r()

Returns:

The only difference between these functions is the way they indicate errors:

SyncSemWait()

If an error occurs, the function returns -1 and sets errno. Any other value returned indicates success (the semaphore was successfully decremented).

SyncSemWait_r()

Returns EOK on success (the semaphore was successfully decremented). This function does NOT set errno. If an error occurs, the function returns one of the values listed in the Errors section.

Errors:

EAGAIN Call would have blocked and try was nonzero.
EDEADLK A deadlock condition was detected.
EINTR A signal interrupted this function.
EINVAL The sync argument doesn’t refer to a valid semaphore.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

sem_destroy(), sem_init(), sem_post(), sem_timedwait(),
sem_trywait(), sem_wait(), SyncDestroy(), SyncSemPost(),
SyncTypeCreate()
Create a synchronization object

**Synopsis:**
```
#include <sys/neutrino.h>

int SyncTypeCreate(
    unsigned type,
    sync_t * sync,
    const struct _sync_attr_t * attr);

int SyncTypeCreate_r(
    unsigned type,
    sync_t * sync,
    const struct _sync_attr_t * attr);
```

**Arguments:**
- `type` One of the following:
  - `_NTO_SYNC_MUTEX_FREE` — create a mutex.
  - `_NTO_SYNCSEM` — create a semaphore.
  - `_NTO_SYNC_COND` — create a condition variable.
- `sync` A pointer to a `sync_t` that the kernel sets up for the synchronization object; see below.
- `attr` A pointer to a `_sync_attr_t` structure that specifies attributes for the object. This structure contains at least the following members:
  - `int protocol` — `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`.

If `attr` is NULL, the default attributes `PTHREAD_PRIO_INHERIT` are assumed.

**Library:** libc

Use the `/-1 c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `SyncTypeCreate()` and `SyncTypeCreate_r()` kernel calls create a synchronization object in the kernel and initializes `sync` for use in other synchronization kernel calls. The synchronization object is local to the process.

These functions are similar, except for the way they indicate errors. See the Returns section for details.

Synchronization objects can be used for mutexes, semaphores, or condition variables.

Don’t call `SyncTypeCreate()` directly; instead, use the POSIX synchronization objects (see `pthread_cond_init()`, `pthread_mutex_init()`, `pthread_rwlock_init()`, and `sem_init()`).

The `sync` argument contains at least the following members:

- `int count` The count for recursive mutexes and semaphores. The kernel sets this member when it creates the synchronization object.
- `int owner` When a mutex is created, this member holds the thread ID of the thread that acquired the mutex. When unowned, the value is 0. It’s set to zero when the synchronization object is created.

The current state of `sync` is summarized below:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Owner</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>-2</td>
<td>Destroyed mutex</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>Statically initialized; auto-created when used</td>
</tr>
</tbody>
</table>

continued…
<table>
<thead>
<tr>
<th>Counter</th>
<th>Owner</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Unlocked mutex</td>
</tr>
<tr>
<td><em>count</em></td>
<td>&gt;0</td>
<td>Recursive counter number of the mutex</td>
</tr>
<tr>
<td><em>count</em></td>
<td>&lt; -1</td>
<td>If the high bit of <em>count</em> is set, it’s a flag meaning “others waiting”</td>
</tr>
<tr>
<td>–</td>
<td>-256</td>
<td>Mutex is dead, waits for revival</td>
</tr>
</tbody>
</table>

The synchronization object is destroyed by a call to `SyncDestroy()`.

**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

`SyncTypeCreate()`

If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

`SyncTypeCreate_r()`

Returns EOK on success. This function does **NOT** set `errno`. If an error occurs, the function can return any value in the Errors section.

**Errors:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>All kernel synchronization objects are in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access <code>sync</code> or <code>attr</code>.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Either</td>
</tr>
</tbody>
</table>
SyncTypeCreate(),
SyncTypeCreate_r()

- the type isn’t one of _NTO_SYNC_COND,
  _NTO_SYNC_MUTEX_FREE or _NTO_SYNC_SEM
Or:
- if the type is correct, and the synchronization object is:
  - a mutex — the protocol isn’t one of
    PTHREAD_PRIO_INHERIT or
    PTHREAD_PRIO_PROTECT.
  - a mutex and PTHREAD_PRIO_PROTECT is specified — the ceiling priority isn’t within the
    kernel priority range.
  - a condvar — the clock type is invalid.
  - a semaphore — the semaphore value exceeds
    SEM_VALUE_MAX.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_init(), pthread_mutex_init(), pthread_rwlock_init(),
sem_init(), SyncCondvarSignal(), SyncCondvarWait(), SyncDestroy(),
SyncMutexLock(), SyncMutexUnlock()

“Synchronization services” in the QNX Neutrino Microkernel chapter of the System Architecture guide
Return the value of a configurable system limit

**Synopsis:**

```c
#include <unistd.h>
#include <limits.h>

long sysconf( int name );
```

**Arguments:**

`name` The name of the limit that you want to get; see below.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `sysconf()` function returns the value of a configurable system limit indicated by `name`.

Configurable limits are defined in `<conffile>`, and contain at least the following values:

- `_SC_ARG_MAX`
  - Maximum length of arguments for the `exec*()` functions, in bytes, including environment data.

- `_SC_CHILD_MAX`
  - Maximum number of simultaneous processes per real user ID.

- `_SC_CLK_TCK`
  - The number of intervals per second used to express the value in type `clock_t`.

- `_SC_NGROUPS_MAX`
  - The maximum number of simultaneous supplementary group IDs per process.
sysconf()

_SC_OPEN_MAX
   Maximum number of files that one process can have open at
   any given time.

_SC_JOB_CONTROL
   If this variable is defined, then job control is supported.

_SC_SAVED_IDS
   If this variable is defined, then each process has a saved set-user
   ID and a saved set-group ID.

_SC_VERSION
   The current POSIX version that is currently supported. A value
   of 198808L indicates the August (08) 1988 standard, as
   approved by the IEEE Standards Board.

Returns:
   The requested configurable system limit. If name isn’t defined for the
   system, -1 is returned.

Examples:

   #include <stdio.h>
   #include <stdlib.h>
   #include <limits.h>
   #include <unistd.h>

   int main( void )
   {
      printf( "\_SC\_ARG\_MAX = \%ld\n\",
              sysconf( \_SC\_ARG\_MAX ) );
      return EXIT\_SUCCESS;
   }

Classification:
   POSIX 1003.1
sysconf()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

confstr(), errno, pathconf()

getconf in the Utilities Reference

Understanding System Limits chapter of the Neutrino User's Guide
sysctl()
Get or set information about the socket manager

Synopsis:
#include <sys/param.h>
#include <sys/sysctl.h>

int sysctl( int * name,
            u_int namelen,
            void * oldp,
            size_t * oldlenp,
            void * newp,
            size_t newlen );

Arguments:

name An array of integers that specifies the Management
      Information Base (MIB) stylename of the item that you
      want to set or get; see below.

namelen The length of the name.

oldp NULL, or a pointer to a buffer where the function can
       store the old value.

oldlenp NULL, or a pointer to a location that initially specifies
       the size of the oldp buffer. The function changes the
       value in this location to be the size of the old information
       stored in the oldp buffer

newp NULL, or a pointer to a buffer that holds the new value.

newlen The size of the new value.

Library:

libsocket

Use the -l socket option to gcc to link against this library.
Description:

The `sysctl()` function retrieves information about the socket manager and allows processes with appropriate privileges to set this information. The data available from `sysctl()` consists of integers and tables. You can also get or set data using the `sysctl` utility at the command line.

The state is described using a Management Information Base (MIB) stylename, specified in `name`, which is a `namelen` length array of integers.

The information is copied into the buffer specified by `oldp`. The size of the buffer is given by the location specified by `oldlenp` before the call, and that location gives the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call delivers as much data as fits in the buffer provided and returns with the error code ENOMEM. If you don’t need the old value, you can set `oldp` and `oldlenp` to NULL.

You can determine the size of the available data by calling `sysctl()` with a NULL parameter for `oldp`. The function stores the size of the available data in the location pointed to by `oldlenp`. For some operations, the amount of space may change often. For these operations, the system attempts to round up, so that the returned size is large enough for a call to return the data shortly thereafter.

To specify a new value, set `newp` to point to a buffer of length `newlen` from which the requested value is to be taken. If you’re not setting a new value, set `newp` to NULL and `newlen` to 0.

The top-level names are defined with a CTL_ prefix in `<sys/sysctl.h>`. QNX 4 supports CTL_NET only. The next and subsequent levels down are found in the following header files:

<table>
<thead>
<tr>
<th>This header file</th>
<th>Contains definitions for</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sys/sysctl.h&gt;</code></td>
<td>Top-level identifiers</td>
</tr>
</tbody>
</table>

continued…
This header file Contains definitions for

<table>
<thead>
<tr>
<th>Header File</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sys/socket.h&gt;</code></td>
<td>Second-level network identifiers</td>
</tr>
<tr>
<td><code>&lt;netinet/in.h&gt;</code></td>
<td>Third-level Internet identifiers and fourth-level IP identifiers</td>
</tr>
<tr>
<td><code>&lt;netinet/icmp_var.h&gt;</code></td>
<td>Fourth-level ICMP identifiers</td>
</tr>
<tr>
<td><code>&lt;netinet/tcp_var.h&gt;</code></td>
<td>Fourth-level TCP identifiers</td>
</tr>
<tr>
<td><code>&lt;netinet/udp_var.h&gt;</code></td>
<td>Fourth-level UDP identifiers</td>
</tr>
</tbody>
</table>

The following code fragment checks whether the UDP packets checksum is enabled:

```c
int mib[5], val;
size_t len;

mib[0] = CTL_NET;
mib[1] = AF_INET;
mib[2] = IPPROTO_UDP;
mib[3] = UDPCTL_CHECKSUM;
len = sizeof(val);
sysctl(mib, 4, &val, &len, NULL, 0);
```

**CTL_NET**

The table and integer information available for the CTL_NET level is detailed below. The Changeable column shows whether a process with appropriate privilege may change the value.

<table>
<thead>
<tr>
<th>Second-level name</th>
<th>Type</th>
<th>Changeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF_INET</td>
<td>internet values</td>
<td>yes</td>
</tr>
</tbody>
</table>

**PF_INET**

PF_INET gets or sets global information about internet protocols.

The third-level name is the protocol. The fourth-level name is the variable name. Here are the currently defined protocols and names:
### Protocol name | Variable name            | Type  | Changeable
---|-------------------------|-------|--------
ip | forwarding             | Integer| Yes   
     | redirect               | Integer| Yes   
     | ttl                    | Integer| Yes   
     | forwsrctxt             | Integer| Yes   
     | directed-broadcast     | Integer| Yes   
     | allowsrcrt             | Integer| Yes   
     | subnetsarelocal        | Integer| Yes   
     | mtudisc                | Integer| Yes   
     | maxfragpackets         | Integer| Yes   
     | sourcecheck            | Integer| Yes   
     | sourcecheck_logint     | Integer| Yes   
icmp | maskrepl               | Integer| Yes   
tcp | rfc1323                 | Integer| Yes   
     | sendspace              | Integer| Yes   
     | recvspace              | Integer| Yes   
     | mssdflt                | Integer| Yes   
     | syn_cache_limit        | Integer| Yes   
     | syn_bucket_limit       | Integer| Yes   
     | syn_cache_interval     | Integer| Yes   
udp | checksum               | Integer| Yes   
     | sendspace              | Integer| Yes   
     | recvspace              | Integer| Yes   

The variables are as follows:
sysctl()

ip.forwarding
Returns 1 when IP forwarding is enabled for the host, meaning that the host is acting as a router.

ip.redirect
Returns 1 when ICMP redirects may be sent by the host. This option is ignored unless the host is routing IP packets. Normally, this option should be enabled on all systems.

ip.ttl
The maximum time-to-live (hop count) value for an IP packet sourced by the system. This value applies to normal transport protocols, not to ICMP.

ip.forwsrcrt
Returns 1 when forwarding of source-routed packets is enabled for the host. This value may be changed only if the kernel security level is less than 1.

ip.directed-broadcast
Returns 1 if directed-broadcast behavior is enabled for the host.

ip.allowsrcrt
Returns 1 if the host accepts source-routed packets.

ip.subnetsarelocal
Returns 1 if subnets are to be considered local addresses.

ip.mtudisc
Returns 1 if path MTU discovery is enabled.

ip.maxfragpackets
Returns the maximum number of fragmented IP packets in the IP reassembly queue.

ip.sourcecheck
Returns 1 if source check for received packets is enabled.
ip.sourcecheck_logint
  Returns the time interval when IP source address verification messages are logged. A value of zero disables the logging.

icmp.maskrepl
  Returns 1 if ICMP network mask requests are to be answered.

tcp.rfc1323
  Returns 1 if RFC1323 extensions to TCP are enabled.

tcp.sendspace
  Returns the default TCP send buffer size.

tcp.recvspace
  Returns the default TCP receive buffer size.

tcp.mssdflt
  Returns the default TCP maximum segment size.

tcp.syn_cache_limit
  Returns the maximum number of entries allowed in the TCP compressed state engine.

tcp.syn_bucket_limit
  Returns the maximum number of entries allowed per hash bucket in the TCP compressed state engine.

tcp.syn_cache_interval
  Returns the TCP compressed state engine’s timer interval.

udp.checksum
  Returns 1 when UDP checksums are being computed and checked.
Disabling UDP checksums is strongly discouraged.

**`udp.sendspace`**

Returns the default UDP send buffer size.

**`udp.recvspace`**

Returns the default UDP receive buffer size.

**Returns:**

0 Success.

-1 An error occurred (errno is set).

**Errors:**

- **EFAULT** The buffers: name, oldp, newp, or the length pointer oldlenp contains an invalid address.

- **EINVAL** The name array is less than two or greater than CTL_MAXNAME; or a non-NULL newp is given and its specified length in newlen is too large or too small.

- **ENOMEM** The length pointed to by oldlenp is too short to hold the requested value.

- **ENOTDIR** The name array specifies an intermediate rather than terminal name.

- **EOPNOTSUPP** The name array specifies an unknown value.

- **EPERM** An attempt was made to set a read-only value; a process, without appropriate privilege, attempts to set or change a value protected by the current system security level.
Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ROUTE protocol

`sysctl` in the *Utilities Reference*
syslog()

Write a message to the system log

Synopsis:
#include <syslog.h>

void syslog( int priority,
            const char * message,
            ...)

Arguments:

priority The priority of the message; see “Message levels,” below.

message The message that you want to write. This message is identical to a printf()-format string, except that \%m is replaced by the current error message (as denoted by the global variable errno). A trailing newline is added if none is present.

The formatting characters that you use in the message determine any additional arguments.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The syslog() function writes message to the system message logger. The message is then written to the system console, log files, and logged-in users, or forwarded to other machines as appropriate. (See the syslogd command.)

The vsyslog() function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of <stdarg.h>.
**Message levels**

The message is tagged with *priority*. Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from the following list (ordered from high to low):

- **LOG_EMERG**: A panic condition. This is normally broadcast to all users.
- **LOG_ALERT**: A condition that should be corrected immediately, such as a corrupted system database.
- **LOG_CRIT**: Critical conditions (for example, hard device errors).
- **LOG_ERR**: General errors.
- **LOG_WARNING**: Warning messages.
- **LOG_NOTICE**: Conditions that aren’t error conditions, but should possibly be specially handled.
- **LOG_INFO**: Informational messages.
- **LOG_DEBUG**: Messages that contain information normally of use only when debugging a program.

**Examples:**

```c
syslog(LOG_ALERT, "who: internal error 23");
openlog("ftpd", LOG_PID, LOG_DAEMON);
setlogmask(LOG_UPTO(LOG_ERR));
syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```
Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

closelog(), openlog(), setlogmask(), vsyslog()

logger, syslogd in the Utilities Reference
sysmgr_reboot()

Reboot the system

Synopsis:

```c
#include <sys/sysmgr.h>

int sysmgr_reboot( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `sysmgr_reboot()` function reboots the calling computer. You need to be `root` for this function to succeed.

Returns:

The `sysmgr_reboot()` function doesn’t return if successful. If an error occurs, it returns:

```
EPERM You need to be root to call this function.
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
sysmgr_reboot()

See also:

procnto in the Utilities Reference
SYSPAGE_CPU_ENTRY() © 2007, QNX Software Systems GmbH & Co. KG.

Return a CPU-specific entry from the system page

Synopsis:

```c
#include <sys/syspage.h>

#define SYSPAGE_CPU_ENTRY( cpu, entry )...
```

Arguments:

- `cpu` The CPU to get the entry for.
- `entry` The entry to get; see below.

Library:

- `libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SYSPAGE_CPU_ENTRY()` macro returns a pointer to the specified `entry` from the part of the system page that’s specific to the given `cpu`.

The best way to reference the system page is via the kernel calls and POSIX cover functions. If there isn’t a function to access the information you need, use `SYSPAGE_CPU_ENTRY()` instead of referencing the `_syspage_ptr` variable directly. For information in the rest of the `syspage_entry` structure, use `SYSPAGE_ENTRY()`.

The only entry you might currently need to use is:

- `ppc.kerinfo`

This structure, defined in `<ppc/syspage.h>`, contains at least the following members:

- `unsigned long pretend_cpu` — we can pretend the chip is this Processor Version Register.
- `unsigned long init_msr` — the initial Machine Status Register for thread creation.
SYSPAGE_CPU_ENTRY()

Returns:
A pointer to the structure for the given entry.

Examples:
```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syspage.h>

int main( void )
{
    printf ("We're pretending to be a type %ld PPC\n",
            SYSPAGE_CPU_ENTRY(ppc,kerinfo)->pretend_cpu);

    return EXIT_SUCCESS;
}
```

Classification:
QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:
SYSPAGE_CPU_ENTRY() is a macro.

See also:
ClockCycles(), SYSPAGE_ENTRY(), _syspage_ptr
Customizing Image Startup Programs chapter of the Building Embedded Systems guide
SYSPAGE_ENTRY() © 2007, QNX Software Systems GmbH & Co. KG.

Return an entry from the system page

Synopsis:

```c
#include <sys/syspage.h>

#define SYSPAGE_ENTRY(entry)...
```

Arguments:

`entry` The entry to get; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `SYSPAGE_ENTRY()` macro returns a pointer to the specified `entry` in the system page.

The best way to reference the system page is via the kernel calls and POSIX cover functions. If there isn’t a function to access the information you need, use `SYSPAGE_ENTRY()` instead of referencing the `_syspage_ptr` variable directly. For information in the CPU-specific part of the `syspage_entry` structure, use `SYSPAGE_CPU_ENTRY()`.

Currently, the only `entry` you’re likely to access with `SYSPAGE_ENTRY()` is:

`qtime` QNX-specific time information. The `qtime_entry` structure contains at least the following members:

- `unsigned long boot_time` — the time, in seconds, since the Unix Epoch (00:00:00 January 1, 1970 Coordinated Universal Time (UTC)) when this system was booted.
- `uint64_t cycles_per_sec` — the number of CPU clock cycles per second for this system. For more information, see `ClockCycles()`.
SYSPAGE_ENTRY()

Returns:

A pointer to the structure for the given entry.

Examples:

```c
#include <sys/neutrino.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syspage.h>

int main( void )
{
    uint64_t cps, cycle1, cycle2, ncycles;
    float sec;

    /* snap the time */
    cycle1=ClockCycles( );

    /* snap the time again */
    cycle2=ClockCycles( );
    ncycles=cycle2-cycle1;
    printf("%lld cycles elapsed \n", ncycles);

    /* find out how many cycles per second */
    cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    printf("This system has %lld cycles/sec.\n", cps);
    sec=(float)ncycles/cps;
    printf("The cycles in seconds is %f \n", sec);

    return EXIT_SUCCESS;
}
```

Classification:

QNX Neutrino

Safety

- Cancellation point: No
- Interrupt handler: Yes
- Signal handler: Yes

continued...
**SYSPAGE_ENTRY()**

| Safety |  
|--------|---
| Thread | Yes |

**Caveats:**

`SYSPAGE_ENTRY()` is a macro.

**See also:**

`ClockCycles()`, `SYSPAGE_CPU_ENTRY()`, `_syspage_ptr`

Customizing Image Startup Programs chapter of the *Building Embedded Systems* guide
Synopsis:
#include <sys/syspage.h>

struct syspage_entry *syspage_ptr

Description:
This global variable holds a pointer to the system page, which
contains information about the system, including the processor type,
bus type, and the location and size of available system RAM. For
information about this structure, see the Customizing Image Startup
Programs chapter of Building Embedded Systems.

The best way to reference the system information page is via the
kernel calls and POSIX cover functions. If there isn’t a function to
access the information you need, you should use the
SYSPAGE_ENTRY() and SYSPAGE_CPU_ENTRY() macros instead
of referencing the syspage_ptr variable directly.

Classification:
QNX Neutrino

See also:
SYSPAGE_CPU_ENTRY(), SYSPAGE_ENTRY()
system()

Execute a system command

Synopsis:

```c
#include <stdlib.h>

int system( const char *command );
```

Arguments:

- `command` NULL, or the system command that you want to execute; see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The behavior of the `system()` function depends on the value of its `command` argument:

- If `command` is NULL, `system()` determines whether or not a shell is present.
- If `command` isn’t NULL, `system()` invokes a copy of the shell, and passes the string `command` to it for processing. This function uses `spawnlp()` to load a copy of the shell.
The shell used is always `/bin/sh`, regardless of the setting of the
SHELL environment variable, because applications may rely on
features of the standard shell, and may fail as a result of running a
different shell.

This means that any command that can be entered to the OS can be
executed, including programs, QNX Neutrino commands, and shell
scripts. The `exec*()` and `spawn*()` functions can only cause programs
to be executed.

Returns:

- If `command` is NULL, `system()` returns zero if the shell isn’t
  present, or a nonzero value if the shell is present.

- If `command` isn’t NULL, `system()` returns the result of invoking a
copy of the shell. If the shell couldn’t be loaded, `system()` returns
  -1; otherwise, it returns the status of the specified command. Use
  the `WEXITSTATUS()` macro to determine the low-order 8 bits of
  the termination status of the process.

For example, assume that `status` is the value returned by `system()`. If
`WEXITSTATUS( status )` returns 255, either the specified
command returned a termination status of 255, or the shell didn’t
exit (i.e. it died from a signal or couldn’t be started at all) and the
return value was 255 due to implementation details. For example,
under QNX Neutrino and most Unix systems, the value is 255 if
`status` is -1, which indicates that the shell couldn’t be executed.

`WEXITSTATUS()` is defined in `<sys/wait.h>`.

For information about macros that extract information from the value
returned by `system()`, see “Status macros” in the description of `wait()`.

When an error has occurred, `errno` contains a value that indicates the
type of error that has been detected.
system()

Examples:

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int main( void )
{
    int rc;

    rc = system( "ls" );
    if( rc == -1 ) {
        printf( "shell could not be run\n" );
    } else {
        printf( "result of running command is %d\n",
                WEXITSTATUS( rc ) );
    }
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abort(), atexit(), close(), errno, execl(), execlp(), execle(),
exefile(), execv(), execve(), execvp(), exit(), _exit(), getenv(),
main(), putenv(), sigaction(), signal(), spawn(), spawnl(),
spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnvp(), spawnve(),
spawnvpe(), wait(), waitpid()
The functions and macros in the C library are described here in alphabetical order:

<table>
<thead>
<tr>
<th>Volume</th>
<th>Range</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A to E</td>
<td>abort() to expm1f()</td>
</tr>
<tr>
<td>2</td>
<td>F to H</td>
<td>fabs() to hypotf()</td>
</tr>
<tr>
<td>3</td>
<td>I to L</td>
<td>ICMP to ltrunc()</td>
</tr>
<tr>
<td>4</td>
<td>M to O</td>
<td>main() to outle32()</td>
</tr>
<tr>
<td>5</td>
<td>P to R</td>
<td>pathconf() to ruserok()</td>
</tr>
<tr>
<td>6</td>
<td>S</td>
<td>sbrk() to system()</td>
</tr>
<tr>
<td>7</td>
<td>T to Z</td>
<td>tan() to ynf()</td>
</tr>
</tbody>
</table>
**tan(), tanf()**

Calculate the tangent of an angle

**Synopsis:**
```
#include <math.h>

double tan( double x );

float tanf( float x );
```

**Arguments:**
- `x` The angle, in radians, for which you want to compute the tangent.

**Library:**
- `libm`

Use the `-l m` option to `qcc` to link against this library.

**Description:**
These functions compute the tangent (specified in radians) of `x`. A large magnitude argument may yield a result with little or no significance.

**Returns:**
The tangent value. If `x` is NAN or infinity, NAN is returned.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

**Examples:**
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( void )
{
```
```c
printf( "%f\n", tan(.5) );

return EXIT_SUCCESS;
}
```

produces the output:

```
0.546302
```

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See also:

`atan(), atan2(), cos(), sin()`
\textit{tanh()}, \textit{tanhf()}

\textit{Calculate the hyperbolic tangent}

\textbf{Synopsis:}
\begin{verbatim}
#include <math.h>

double tanh( double x );

float tanhf( float x );
\end{verbatim}

\textbf{Arguments:}
\begin{verbatim}
x  The angle, in radians, for which you want to compute the hyperbolic tangent.
\end{verbatim}

\textbf{Library:}
\begin{verbatim}
libm
\end{verbatim}

Use the -l m option to qcc to link against this library.

\textbf{Description:}
These functions compute the hyperbolic tangent (specified in radians) of \( x \).

When the \( x \) argument is large, partial or total loss of significance may occur.

\textbf{Returns:}
The hyperbolic tangent value.

\begin{center}
\begin{tabular}{l}
\textbf{If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set errno to 0, call the function, and then check errno again. These functions don’t change errno if no errors occurred.}
\end{tabular}
\end{center}

\textbf{Examples:}
\begin{verbatim}
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
\end{verbatim}
int main( void )
{
    printf( "%f\n", tanh(.5) );
    return EXIT_SUCCESS;
}

produces the output:

0.462117

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

cosh(), errno, sinh()
**Synopsis:**

```c
#include <termios.h>

int tcdrain(int fildes);
```

**Arguments:**

- `fildes`: A file descriptor that’s associated with the device that you want to wait for.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `tcdrain()` function waits until all output has been physically transmitted to the device associated with `fildes`, or until a signal is received.

**Returns:**

- 0  Success.
- -1  An error occurred (`errno` is set).

**Errors:**

- E_BADF  The argument `fildes` is invalid.
- EINTR  A signal interrupted the operation.
- E_NOSYS  The resource manager associated with `fildes` doesn’t support this call.
- ENOTTY  The argument `fildes` doesn’t refer to a terminal device.
Examples:

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fildes;

    fildes = open( "\dev/ser1", O_RDWR );
    write( fildes, "ATH", 3 );

    /* Wait for data to transmit before returning */
    tcdrain( fildes );
    close( fildes );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`write()`
**Synopsis:**

```c
#include <termios.h>

int tcdropline( int fd,  
                int duration );
```

**Arguments:**

*fd*  
A file descriptor that’s associated with the line that you want to disconnect.

*duration*  
The number of milliseconds that you want to drop the line for.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `tcdropline()` function initiates a disconnect condition on the communication line associated with the opened file descriptor indicated by *fd*.

The disconnect condition lasts at least *duration* milliseconds, or approximately 300 milliseconds if *duration* is zero. The system rounds the effective value of *duration* up to the next highest supported interval, which is typically a multiple of 100 milliseconds.

**Returns:**

0  
Success.

-1  
An error occurred (*errno* is set).
tcdropline()

Errors:

EBADF       Invalid fd argument.
ENOSYS      The resource manager associated with fd doesn’t support this call.
ENOTTY      The argument fd doesn’t refer to a terminal device.

Examples:

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    fd = open( "/dev/ser1", O_RDWR );

    /* Disconnect for 500 milliseconds */
tcdropline( fd, 500 );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

QNX 4

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

tcdrain(), tcflow(), tcflush()
Perform a flow-control operation on a data stream

Synopsis:

```
#include <termios.h>

int tcflow(int fildes, int action);
```

Arguments:

- `fildes` A file descriptor that’s associated with the data stream that you want to perform the operation on.
- `action` The action you want to perform; see below.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcflow()` function performs a flow-control operation on the data stream associated with `fildes`, depending on the values in `action`.

At least the following actions are defined in `<termios.h>`:

- `TCOOFF` Use software flow control to suspend output on the device associated with `fildes`.
- `TCOFFHW` Use hardware flow control to suspend output on the device associated with `fildes`.
- `TCOON` Use software flow control to resume output on the device associated with `fildes`.
- `TCOONHW` Use hardware flow control to resume output on the device associated with `fildes`.
- `TCIOFF` Cause input to be flow-controlled by sending a STOP character immediately across the
communication line associated with \textit{fildes}, (that is, software flow control).

TCIOFFHW Cause input to be flow-controlled by using hardware control.

TCION Resume input by sending a START character immediately across the communication line associated with \textit{fildes} (that is, software flow control).

TCIONHW Cause input to be resumed by using hardware flow control.

Returns:

0 Success.
-1 An error occurred (\textit{errno} is set).

Errors:

EBADF Invalid \textit{fildes} argument.
EINVAL Invalid \textit{action} argument.
ENOSYS The resource manager associated with \textit{fildes} doesn’t support this call.
ENOTTY The argument \textit{fildes} doesn’t refer to a terminal device.

Examples:

\begin{verbatim}
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    fd = open( "/dev/ser1", O_RDWR );

    return fd;
}
\end{verbatim}
/* Resume output on flow-controlled device */
tcflow( fd, TCOON );

close( fd );
return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

tcdrain(), tcflush(), tcsendbreak()


\textbf{Synopsis:}

\begin{verbatim}
#include <termios.h>

int tcflush( int fildes,  
             int queue_selector );
\end{verbatim}

\section*{Arguments:}

\begin{itemize}
  \item \textit{fildes} \hspace{1cm} A file descriptor that’s associated with the data stream that you want to perform the operation on.
  \item \textit{queue_selector} \hspace{1cm} The stream or streams that you want to flush. At least the following values for \textit{queue_selector} are defined in <termios.h>:
    \begin{itemize}
      \item TCIFLUSH — discard all data that’s received, but not yet read, on the device associated with \textit{fildes}.
      \item TCOFLUSH — discard all data that’s written, but not yet transmitted, on the device associated with \textit{fildes}.
      \item TCIOFLUSH — discard all data that’s written, but not yet transmitted, as well as all data that’s received, but not yet read, on the device associated with \textit{fildes}.
    \end{itemize}
\end{itemize}

\section*{Library:}

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

\section*{Description:}

The \texttt{tcflush()} functionFlushes the input stream, the output stream, or both, depending on the value of the argument \textit{queue_selector}. 
tcflush()

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EBADF Invalid fildes argument.

EINVAL Invalid queue_selector argument.

ENOSYS The resource manager associated with fildes doesn’t support this call.

ENOTTY The argument fildes doesn’t refer to a terminal device.

Examples:

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fildes;

    fildes = open( "/dev/ser1", O_RDWR );

    /* Throw away all input data */
    tcflush( fildes, TCI_FLUSH );

    close( fildes );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1
tcflush()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

tcdrain(), tcflow(), tcsendbreak()
tcgetattr()
Get the current terminal control settings

Synopsis:

```c
#include <termios.h>

int tcgetattr( int fildes,
               struct termios *termios_p);
```

Arguments:

- `fildes` — The file descriptor associated with the terminal device.
- `termios_p` — A pointer to a `termios` structure in which `tcgetattr()` can store the terminal’s control attributes.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcgetattr()` function gets the current terminal control settings for the opened device indicated by `fildes`, and stores the results in the structure pointed to by `termios_p`.

Returns:

- 0 — Success.
- -1 — An error occurred; `errno` is set.

Errors:

- `EBADF` — The `fildes` argument is invalid.
- `ENOSYS` — The resource manager associated with `fildes` doesn’t support this call.
- `ENOTTY` — The `fildes` argument doesn’t refer to a terminal device.
Examples:

See tcsetattr().

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fpathconf(), tcsetattr(), termios

Chapter 7 of POSIX 1003.1
Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp( int fildes );
```

Arguments:

- `fildes` A file descriptor that’s associated with the device whose process group ID you want to get.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcgetpgrp()` function returns the process group ID of the foreground process that’s associated with the device indicated by `fildes`.

Returns:

The ID of foreground process group. If an error occurs, -1 is returned, and `errno` is set.

Errors:

- EBADF The argument `fildes` is invalid.
- ENOSYS The resource manager associated with `fildes` doesn’t support this call.
- ENOTTY The argument `fildes` isn’t associated with a terminal device.
tcgetpgrp()

Examples:

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf( "STDIN directs breaks to pgrp %d\n", 
            tcgetpgrp( 0 ) );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`signal()`, `tcsetpgrp()`
Synopsis:

```
#include <unistd.h>

int tcgetsid( int filedes );
```

Arguments:

- `filedes` A file descriptor that’s associated with the device whose ID you want to get.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcgetsid()` function returns the process group ID of the session for which the terminal specified by `filedes` is the controlling terminal.

Returns:

The process group ID associated with the terminal, or `-1` if an error occurs (`errno` is set).

Errors:

- EACCES The `filedes` argument isn’t associated with a controlling terminal.
- EBADF The `filedes` argument isn’t a valid file descriptor.
- ENOTTY The file associated with `filedes` isn’t a terminal.
Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

tcsetsid()
Synopsis:

```c
#include <termios.h>

int tcgetsize(int filedes,
               int* prow,
               int* pcol);
```

Arguments:

- `fdedes`: A file descriptor that’s associated with the device whose size you want to get.
- `prows`, `pcols`: NULL, or pointers to locations where the function can store the number of rows and columns.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcgetsize()` function gets the size of the character device associated with `filedes` and stores the number of rows and columns in `prows` and `pcols` if they're not NULL.

Returns:

- `0`: Success.
- `-1`: An error occurred (`errno` is set).

Errors:

- `EACCES`: The `filedes` argument isn’t associated with a controlling terminal.
- `EBADF`: The `filedes` argument isn’t a valid file descriptor.
- `ENOTTY`: The file associated with `filedes` isn’t a terminal.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:


tcsetsize()
Synopsis:

```c
#include <termios.h>

int tcinject( int fd,
              char *buf,
              int n );
```

Arguments:

- `fd` - A file descriptor that’s associated with the device whose input buffer you want to add characters to.
- `buf` - A pointer to a buffer that contains the characters that you want to insert.
- `n` - The number of characters to insert. If `n` is positive, the characters are written to the canonical (edited) queue. If `n` is negative, the characters are written to the raw queue.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcinject()` function injects `n` characters pointed to by `buf` into the input buffer of the device given in `fd`.

Note that while injecting into the canonical queue, editing characters in `buf` are acted upon as though the user entered them directly from the device. If `buf` doesn’t contain a newline (`\n`), carriage return (`\r`) or a data-forwarding character such as an EOF, data doesn’t become available for reading. If `buf` does contain a data-forwarding character, it should contain only one as the last character in `buf`.

This function is useful for implementing command-line recall algorithms by injecting recalled lines into the canonical queue.
tcinject()

Returns:

0 Success.

-1 An error occurred (errno is set).

Errors:

EBADF The fd argument is invalid or the file isn’t opened for read.

ENOSYS This function isn’t supported for the device opened.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

int main( void )
{
    char *p = "echo Hello world!\n";

    /* Inject the line all at once */
    tcinject(0, p, strlen(p));

    /* Inject the line one character at a time */
    while(*p)
        tcinject(0, p++, 1);
    return EXIT_SUCCESS;
}
```

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
</tbody>
</table>

continued…
### Safety

<table>
<thead>
<tr>
<th>Thread</th>
<th>Yes</th>
</tr>
</thead>
</table>

© 2007, QNX Software Systems GmbH & Co. KG.
**tcischars()**

Determine the number of characters waiting to be read

**Synopsis:**
```c
#include <termios.h>

int tcischars(int fi1deds);
```

**Arguments:**
- `fildes` A file descriptor that’s associated with the device that you want to check.

**Library:**
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `tcischars()` function checks to see how many characters are waiting to be read from the given file descriptor, `fildes`.

**Returns:**
The number of characters waiting to be read, or -1 if an error occurred.

**Errors:**
- `ENOTTY` The `fd` argument isn’t the file descriptor for a character device.

**Classification:**
QNX Neutrino
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Internet Transmission Control Protocol

Synopsis:

```
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_STREAM,
            0 );
```

Description:

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It’s a byte-stream protocol used to support the SOCK_STREAM abstraction.

TCP uses the standard Internet address format and also provides a per-host collection of “port addresses.” Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets using the TCP protocol are either active or passive. Active sockets initiate connections to passive sockets. By default, TCP sockets are created active.

To create a passive socket, you must bind the socket with the `bind()` system call, and then use the `listen()` system call. Only passive sockets may use the `accept()` call to accept incoming connections; only active sockets may use the `connect()` call to initiate connections.

Passive sockets may “underspecify” their location to match incoming connection requests from multiple networks. With this technique, termed wildcard addressing, a single server can provide service to clients on multiple networks. If you wish to create a socket that listens on all networks, the Internet address INADDR_ANY must be bound. You can still specify the TCP port at this time. If the port isn’t specified, the system assigns one.

Once a connection has been established, the socket’s address is fixed by the peer entity’s location. The address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity’s network.
TCP supports several socket options (defined in `<netinet/tcp.h>`) that you can set with `setsockopt()` and retrieve with `getsockopt()`. The option level for these calls is the protocol number for TCP, available from `getprotobyname()`.

TCP_NODELAY
Under most circumstances, TCP sends data when it’s presented. When outstanding data hasn’t yet been acknowledged, TCP gathers small amounts of output to be sent in a single packet once an acknowledgment is received.

For a few clients (such as windowing systems that send a stream of mouse events that receive no replies), this packetization may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY, to defeat this algorithm.

TCP_MAXSEG
The Maximum Segment Size (MSS) for a TCP connection. The value returned is the maximum amount of data that TCP sends to the other end. If this value is fetched before the socket is connected, the value returned is the default value that’s used if an MSS option isn’t received from the other end.

TCP_KEEPALIVE
Specifies the idle time in seconds for the connection before TCP starts sending “keepalive” probes. The default value is 2 hours. This option is effective only when the SO_KEEPALIVE socket option is enabled.

You can use options at the IP transport level with TCP (see the IP protocol. Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

Returns:
A descriptor referencing the socket, or -1 if an error occurs (`errno` is set).
Errors:

EADDRINUSE You tried to create a socket with a port that’s already been allocated.

EADDRNOTAVAIL You tried to create a socket with a network address for which no network interface exists.

ECONNREFUSED The remote peer actively refused connection establishment (usually because no process was listening to the port).

ECONNRESET The remote peer forced the connection to be closed.

EISCONN You tried to establish a connection on a socket that already has one.

ENOBUFS The system ran out of memory for an internal data structure.

ETIMEDOUT A connection was dropped due to excessive retransmissions.

See also:

IP protocol

accept(), bind(), connect(), getprotobyname(), getsockopt(), listen(), setsockopt(), socket()

RFC 793
tcsendbreak()

Assert a break condition over a communications line

Synopsis:

```
#include <termios.h>

int tcsendbreak(int fd, int duration);
```

Arguments:

- `fd` A file descriptor that's associated with the line that you want to assert the break on.
- `duration` The number of milliseconds that you want to break for.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcsendbreak()` function asserts a break condition over the communication line associated with the opened file descriptor indicated by `fd`.

The break condition lasts for at least `duration` milliseconds, or approximately 300 milliseconds if `duration` is zero. The system rounds the effective value of `duration` up to the next highest supported interval, which is typically a multiple of 100 milliseconds.

Returns:

- `0` Success.
- `-1` An error occurred (`errno` is set).
Errors:

EBADF The argument *fildes* is invalid.

ENOSYS The resource manager associated with *fildes* doesn’t support this call.

ENOTTY The argument *fildes* doesn’t refer to a terminal device.

Examples:

```c
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    int fd;

    fd = open( "/dev/ser1", O_RDWR );

    /* Send a 500 millisecond break */
    tcsendbreak( fd, 500 );

    close( fd );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`tcdrain()`, `tcflow()`, `tcflush()`
**Synopsis:**

```c
#include <termios.h>

int tcsetattr( int fildes,
               int optional_actions,
               const struct termios *termios_p );
```

**Arguments:**

- `fildes` The file descriptor associated with the terminal device.
- `termios_p` A pointer to a `termios` structure that describes the attributes that you want to set for the terminal device.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `tcsetattr()` function sets the current terminal control settings for the opened device indicated by `fildes` to the values stored in the structure pointed to by `termios_p`.

The operation of `tcsetattr()` depends on the values in `optional_actions`:

- **TCSANOW** The change is made immediately.
- **TCSADRAIN** No change is made until all currently written data has been transmitted.
- **TCSAFLUSH** No change is made until all currently written data has been transmitted, at which point any received but unread data is also discarded.

The `termios` control structure is defined in `<termios.h>`. For more information, see `tcgetattr()`.
tcsetattr()

Returns:

0      Success.
-1     An error occurred; errno is set.

Errors:

EBADF   The argument fildes is invalid;
EINVVAL The argument action is invalid, or one of the members
of termios_p is invalid.
ENOSYS  The resource manager associated with fildes doesn’t
support this call.
ENOTTY  The argument fildes doesn’t refer to a terminal device.

Examples:

```c
#include <stdlib.h>
#include <termios.h>

int raw( int fd )
{
    struct termios termios_p;

    if( tcgetattr( fd, &termios_p ) )
        return( -1 );

    termios_p.c_cc[VMIN] = 1;
    termios_p.c_cc[VTIME] = 0;
    termios_p.c_lflag &= ~( ECHO|ICANON|ISIG|
                           ECHOE|ECHOK|ECHONL );
    termios_p.c_oflag &= ~( OPOST );
    return( tcsetattr( fd, TCSADRAIN, &termios_p ) );
}

int unraw( int fd )
{
    struct termios termios_p;

    if( tcgetattr( fd, &termios_p ) )
        return( -1 );

    termios_p.c_lflag |= ( ECHO|ICANON|ISIG|
                          ECHOE|ECHOK|ECHONL );
```
termios_p.c_oflag |= ( OPOST );
return( tcsetattr( fd, TCSADRAIN, &termios_p ) );
}

int main( void )
{
    raw( 0 );
    /*
     * Stdin is now "raw"
     */
    unraw ( 0 );
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

errno, select(), tcgetattr(), termios
tcsetpgrp()  
Set the process group ID for a device

Synopsis:
#include <sys/types.h>
#include <unistd.h>

int tcsetpgrp( int fildes,
              pid_t pgrp_id );

Arguments:

fildes  A file descriptor that’s associated with the device whose
        process group ID you want to set.

pgrp_id  The process group ID that you want to assign to the
         device.

Library:

libc
Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:

The tcsetpgrp() function sets the process group ID associated with the
device indicated by fildes to be pgrp_id.

If successful, the tcsetpgrp() function causes subsequent breaks on the
indicated terminal device to generate a SIGINT on all process in the
given process group.

Returns:

0    Success.

-1    An error occurred (errno is set).
**tcsetpgrp()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Errors:**

- **EBADF**  
  The argument *fildes* is invalid.

- **EINVAL**  
  The argument *pgrp_id* is invalid.

- **ENOSYS**  
  The resource manager associated with *fildes* doesn’t support this call.

- **ENOTTY**  
  The argument *fildes* isn’t associated with a terminal device.

- **EPERM**  
  The argument *pgrp_id* isn’t part of the same session as the calling process.

**Examples:**

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
  /*
   * Direct breaks on stdin to me
   */
  tcsetpgrp( 0, getpid() );
  return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

3400  C Library — T to Z  September 10, 2007
See also:

`signal()`, `tcgetpgrp()`
Make a terminal device a controlling device

Synopsis:

```
#include <termios.h>

int tcsetsid( int fd,
              pid_t pid );
```

Arguments:

- **fd** A file descriptor that’s associated with the device that you want to make a controlling device.
- **pid** The ID of the process that you want to associate with the controlling device.

Library:

```
libc
```

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcsetsid()` function makes the terminal device associated with the file descriptor argument `fd` into a controlling terminal that’s associated with the process `pid`. If successful, this call causes subsequent hangup conditions on the terminal device `fd` to generate a SIGHUP signal on the given process.

This call is equivalent to calling `ioctl(fd, TIOCSCSCTTY)` to set the controlling terminal to the current process. You can clear the controlling terminal by passing `-1` as `fd`.

Returns:

- **0** Success.
- **-1** Failure; `errno` is set.
Errors:

EBADF     Invalid file descriptor.
EINV AL    The argument pid is invalid.
ENOSYS, ENOTTY
    The argument fd isn’t associated with a terminal device.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ioctl(), tcgetsid()
tcsetsize()

Set the size of a character device

Synopsis:

```c
#include <termios.h>

int tcsetsize( int filedes,
               int rows,
               int cols );
```

Arguments:

- `fd`es A file descriptor that’s associated with the device whose size you want to set.
- `rows`, `cols` The number of rows and columns that you want to use.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tcsetsize()` function sets the size of the character device associated with `fd` to the given number of rows and columns.

Returns:

-0 Success.

-1 An error occurred (`errno` is set).

Errors:

- `EACCES` The `fd` argument isn’t associated with a controlling terminal.
- `EBADF` The `fd` argument isn’t a valid file descriptor.
- `EINVAL` The rows or cols argument is invalid.
- `ENOTTY` The file associated with `fd` isn’t a terminal.
Classification:
QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`tcgetsize()`
**Synopsis:**

```c
#include <unistd.h>

off_t tell( int filedes );

off64_t tell( int filedes );
```

**Arguments:**

- `filedes` The file descriptor of the file whose position you want to get.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `tell()` function determines the current file position for any subsequent `read()` or `write()` operation (that is, any subsequent unbuffered file operation). The `filedes` value is the file descriptor returned by a successful call to `open()`.

You can use the returned value in conjunction with `lseek()` to reset the current file position.

**Returns:**

The current file position, expressed as the number of bytes from the start of the file, or -1 if an error occurs (`errno` is set). A value of 0 indicates the start of the file.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
```
char buffer[] = { "A text record to be written" };

int main( void )
{
    int filedes;
    int size_written;

    /* open a file for output */
    /* replace existing file if it exists */
    filedes = open("file",
                    O_WRONLY | O_CREAT | O_TRUNC,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);

    if( filedes != -1 ) {
        /* print file position */
        printf( "%ld\n", tell( filedes ) );

        /* write the text */
        size_written = write( filedes, buffer,
                             sizeof( buffer ) );

        /* print file position */
        printf( "%ld\n", tell( filedes ) );

        /* close the file */
        close( filedes );
    }
    return EXIT_SUCCESS;
}

produces the output:

0
28

Classification:

tell() is QNX 4; tell64() is Large-file support

Safety

Cancellation point  Yes

continued...
**tell(), tell64()**  

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

**See also:**

chsize(), close(), creat(), dup(), dup2(), eof(), errno, execl(), execle(), execle(), execle(), execlpe(), execv(), execve(), execvp(), execve(), lseek(), open(), read(), sopen(), stat(), umask(), write()


**Synopsis:**

```c
#include <dirent.h>

long int telldir( DIR * dirp );
```

**Arguments:**

- `dirp` The directory stream for which you want to get the current location.

**Library:**

- `libc`

  Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

**Description:**

The `telldir()` function obtains the current location associated with the directory stream specified by `dirp`.

**Returns:**

The current position of the specified directory stream, or -1 if an error occurs (`errno` is set).

**Errors:**

- `EBADF` The `dirp` argument doesn’t refer to an open directory stream.

**Classification:**

- POSIX 1003.1 XSI
# telldir()

## Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

## See also:

- closedir()
- errno
- lstat()
- opendir()
- readdir()
- readdir_r()
- rewinddir()
- seekdir()
- stat()
tempnam()
Create a name for a temporary file

Synopsis:

```c
#include <stdio.h>

char* tempnam( const char* dir, const char* pfx );
```

Arguments:

- `dir`: NULL, or the directory to use in the pathname.
- `pfx`: NULL, or a prefix to use in the pathname.

If `pfx` isn’t NULL, the string it points to must be no more than 5 bytes long.

Library:

```
libc
Use the `-l c` option to qcc to link against this library. This library is usually included automatically.
```

Description:

The `tempnam()` function generates a pathname for use as a temporary file. The pathname is in the directory specified by `dir` and has the prefix specified in `pfx`.

If `dir` is NULL, the pathname is prefixed with the first accessible directory contained in:

- the temporary file directory `P_tmpdir` (defined in `<stdio.h>`)
- the `TMPDIR` environment variable
- the `_PATH_TMP` constant (defined in `<paths.h>`).

If all of these paths are inaccessible, `tempnam()` attempts to use `/tmp` and then the current working directory.

The `tempnam()` function generates up to TMP_MAX unique file names before it starts to recycle them.
tempnam()

Returns:

A pointer to the generated file name, which you should deallocate with the `free()` function when the application no longer needs it, or NULL if an error occurs.

Errors:

ENOMEM There’s insufficient memory available to create the pathname.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

Caveats:

The `tempnam()` functions creates only pathnames; the application must create and remove the files.

It’s possible for another thread or process to create a file with the same name between the time the pathname is created and the file is opened.

See also:

`free()`, `tmpfile()`, `tmpnam()`, `unlink()`
Synopsis:

```c
struct termios {
    tcflag_t  c_iflag;
    tcflag_t  c_oflag;
    tcflag_t  c_cflag;
    tcflag_t  c_lflag;
    cc_t      c_cc [NCCS];
    uint32_t  reserved[3];
    speed_t   c_ispeed;
    speed_t   c_ospeed;
};
```

Description:

The `termios` control structure is defined in `<termios.h>`, and contains at least the members described below.

**tcflag_t c_iflag**

Input modes. This member contains at least the following bits:

- **BRKINT** Signal interrupt on break.
- **ICRNL** Map CR to NL on input.
- **IGNBRK** Ignore break conditions.
- **IGNCR** Ignore CR.
- **IGNPAR** Ignore characters with parity errors.
- **INLCR** Map NL to CR on input.
- **INPCK** Enable input parity check.
- **ISTRIP** Strip top bit from character.
- **IXOFF** Enable software input flow control (via START/STOP chars).
- **IXON** Enable software output flow control (via START/STOP chars).
- **PARMRK** Mark parity errors in the input data stream.

**tcflag_t c_oflag**

Output modes. This member contains at least the following bits:
OPOST  Perform output processing.

\texttt{tcflag_t c\_cflag}

Control modes. This member contains at least the following bits:

\begin{itemize}
  \item \texttt{CLOCAL}  Ignore modem status lines.
  \item \texttt{CREASE}  Enable receiver.
  \item \texttt{CSIZE}  Number of data bits per character.
  \item \texttt{CS5}  5 data bits.
  \item \texttt{CS6}  6 data bits.
  \item \texttt{CS7}  7 data bits.
  \item \texttt{CS8}  8 data bits.
  \item \texttt{CSTOPB}  Two stop bits, else one.
  \item \texttt{HUPCL}  Hang up on last close.
  \item \texttt{IHFLOW}  Support input flow control using the hardware handshaking lines.
  \item \texttt{OHFLOW}  Support output flow control using the hardware handshaking lines.
  \item \texttt{PARENB}  Parity enable.
  \item \texttt{PARODD}  Odd parity, else even.
  \item \texttt{PARSTK}  Stick parity (mark parity if \texttt{PARODD} is set, else space parity).
\end{itemize}

\texttt{tcflag_t c\_lflag}

Local modes. This member contains at least the following bits:

\begin{itemize}
  \item \texttt{ECHO}  Enable echo.
  \item \texttt{ECHOE}  Echo ERASE as destructive backspace.
  \item \texttt{ECHOK}  Echo KILL as a line erase.
  \item \texttt{ECHONL}  Echo `\n`, even if \texttt{ECHO} is off.
  \item \texttt{ICANON}  Canonical input mode (line editing enabled).
\end{itemize}
IEXTEN   QNX extensions to POSIX are enabled.
ISIG     Enable signals.
NOFLSH   Disable flush after interrupt, quit, or suspend.
TOSTOP   Send SIGTTOU for background output.

cc_t c_cc[NCCS]
Control characters. The array c_cc includes at least the following control characters:

- c_cc[VEOF]   EOF character.
- c_cc[VEOL]   EOL character.
- c_cc[VERASE] ERASE character.
- c_cc[VINTR]  INTR character.
- c_cc[VKILL]  KILL character.
- c_cc[VMIN]   MIN value.
- c_cc[VQUIT]  QUIT character.
- c_cc[VSUSP]  SUSP character.
- c_cc[VTIME]  TIME value.
- c_cc[VSTART] START character.
- c_cc[VSTOP]  STOP character.

The following control characters are also defined, but are only acted on if they’re immediately preceded by the nonzero characters in c_cc[VPREFIX][4], and are immediately followed by the nonzero characters in c_cc[VSUFFIX][4] and the IEXTEN bit of c_lflag is set:

- c_cc[VLEFT]  Left cursor motion.
- c_cc[VRIGHT] Right cursor motion.
- c_cc[VUP]    Up cursor motion.
- c_cc[VDOWN]  Down cursor motion.
- c_cc[VINS]   Insert character.
termios

© 2007, QNX Software Systems GmbH & Co. KG.

c_cc[VDEL]       Delete character.
c_cc[VRUB]       Rubout character.
c_cc[VCAN]       Cancel character.
c_cc[VHOME]      Home character.
c_cc[VEND]       End character.

Any of the control characters in the c_cc array can be disabled by setting that character to the _PC_VDISABLE parameter which is returned by fpathconf() (typically a zero).

speed_t c_ispeed

Input baud rate. This member should be queried and set with the cfgetispeed() and cfsetispeed() functions.

speed_t c_ospeed

Output baud rate. This member should be queried and set with the cfgetospeed() and cfsetospeed() functions.

Classification:

POSIX 1003.1

See also:

cfgetispeed(), cfgetospeed(), cfsetispeed(), cfsetospeed(), cfmakeraw(), fpathconf(), forkpty(), openpty(), pathconf(), readcond(), tcgetattr(), tcsetattr()
thread_pool_control()
Control the thread pool behavior

Synopsis:
#include <sys/iofunc.h>
#include <sys/dispatch.h>

thread_pool_control( thread_pool_t * pool,
                      thread_pool_attr_t * attr,
                      uint16_t lower,
                      uint16_t upper,
                      unsigned flags )

Arguments:

pool
A thread pool handle that was returned by thread_pool_create().

attr
A pointer to a thread_pool_attr_t structure that specifies the attributes that you want to use for the thread pool. For more information, see “Thread-pool attributes,” in the documentation for thread_pool_create().

lower, upper
This function blocks until the number of threads created is between the range of upper and lower, unless you set THREAD_POOL_CONTROL_NONBLOCK in flags.

flags
Which attributes you want to change for the thread pool; any combination of the following bits:

- THREAD_POOL_CONTROL_HIWATER — adjust the high-water value of the number of threads allowed in the thread pool.
- THREAD_POOL_CONTROL_INCREMENT — adjust the increment value of the number of threads.
- THREAD_POOL_CONTROL_LOWATER — adjust the low-water value of the number of threads allowed in the thread pool.
thread_pool_control()

- THREAD_POOL_CONTROL_MAXIMUM — adjust the maximum value of the number of threads allowed in the thread pool.
- THREAD_POOL_CONTROL_NONBLOCK — don’t block while creating threads.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

Use thread_pool_control() to specify a thread pool’s behavior and adjust its attributes.

Having several threads call this function with the same thread pool handle isn’t recommended.

Returns:

-1 if an error occurs (errno is set).

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

© 2007, QNX Software Systems GmbH & Co. KG.
See also:

\textit{thread\_pool\_destroy()}, \textit{thread\_pool\_create()}, \textit{thread\_pool\_limits()}, \textit{thread\_pool\_start()}
thread_pool_create()  © 2007, QNX Software Systems GmbH & Co. KG.

Create a thread pool handle

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

thread_pool_t * thread_pool_create (
    thread_pool_attr_t * pool_attr,
    unsigned flags);
```

Arguments:

- **pool_attr**  
  A pointer to a `thread_pool_attr_t` structure that specifies the attributes that you want to use for the thread pool. For more information, see “Thread-pool attributes,” below.

- **flags**  
  Flags (defined in `<sys/dispatch.h>`) that affect what happens to the thread that’s creating the pool:
  - `POOL_FLAG_EXIT_SELF` — when the pool is started using `thread_pool_start()`, exit the thread that called this function.
  - `POOL_FLAG_USE_SELF` — when the pool is started, use the calling thread as part of the pool.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `thread_pool_create()` function creates a thread pool handle. This handle is then used to start a thread pool with `thread_pool_start()`. With the thread pool functions, you can create and manage a pool of worker threads.
How it works

The worker threads work in the following way:

- When a new worker thread is created, a context is allocated, which the thread uses to do its work.

- The thread then calls the blocking function. This function blocks until the thread has work to do. For example, the blocking function could call `MsgReceive()` to wait for a message.

- After the blocking function returns, the worker thread calls the handler function, which performs the actual work.

- When the handler function returns, the thread calls the blocking function again.

The thread continues to block and handle events until the thread pool decides this worker thread is no longer needed. Finally, when the worker thread exits, it releases the allocated context.

The thread pool manages these worker threads so that there’s a certain number of them in the blocked state. Thus, as threads become busy in the handler function, the thread pool creates new threads to keep a minimum number of threads in a state where they can accept requests from clients. By the same token, if the demand on the thread pool goes down, the thread pool lets some of these blocked threads exit.

Thread-pool attributes

The `pool_attr` argument sets the:

- functions that get called, when a new thread is started or one dies, to allocate and free contexts used by threads

- blocking and handler functions

- parameters of the thread pool such as the number of worker threads, etc.

The `thread_pool_attr_t` structure that the `pool_attr` argument points to is defined in `<sys/dispatch.h>` as:
typedef struct _thread_pool_attr {
    THREAD_POOL_HANDLE_T *handle;
    THREAD_POOL_PARAM_T *(*block_func) (THREAD_POOL_PARAM_T *ctp);
    void (*unblock_func) (THREAD_POOL_PARAM_T *ctp);
    int (*handler_func) (THREAD_POOL_PARAM_T *ctp);
    THREAD_POOL_PARAM_T *(*context_alloc) (THREAD_POOL_PARAM_T *handle);
    void (*context_free) (THREAD_POOL_PARAM_T *ctp);
    pthread_attr_t *attr;
    unsigned short lo_water;
    unsigned short increment;
    unsigned short hi_water;
    unsigned short maximum;
    unsigned reserved[8];
} thread_pool_attr_t;

The members include:

handle A handle that gets passed to the context_alloc function.

block_func The function that’s called when the worker thread is ready to block, waiting for work. This function returns a pointer that’s passed to handler_func.

unblock_func The function that’s called to unblock threads. If you use dispatch_block() as the block_func, use dispatch_unblock() as the unblock_func.

handler_func The function that’s called after block_func returns to do some work. The function is passed the pointer returned by block_func.

context_alloc The function that’s called when a new thread is created by the thread pool. It is passed handle. The function returns a pointer, which is then passed to the blocking function, block_func.

context_free The function that’s called when the worker thread exits, to free the context allocated with context_alloc.
thread_pool_create()

attr
A pointer to a pthread_attr_*() function that’s passed to pthread_create(). The pthread_attr_*() functions set the stack size, priority, etc. of the worker threads. If NULL, default values are used.

lo_water
The minimum number of threads that the pool should keep in the blocked state (i.e. threads that are ready to do work).

increment
The number of new threads created at one time.

hi_water
The maximum number of threads to keep in a blocked state.

maximum
The maximum number of threads that the pool can create.

Returns:
A thread pool handle, or NULL if an error occurs (errno is set).

Errors:
ENOMEM Insufficient memory to allocate internal data structures.

Examples:
Here’s a simple multithreaded resource manager:

/* Define an appropriate interrupt number: */
#define INTNUM 0

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>

static resmgr_connect_funcs_t connect_funcs;
static resmgr_iofuncs_t io_funcs;
static iofunc_attr_t attr;
```c
void *interrupt_thread( void *data)
/* *data isn’t used */
{
    struct sigevent event;
    int id;

    /* fill in "event" structure */
    memset( &event, 0, sizeof(event) );
    event.sigev_notify = SIGEV_INTR;

    /* INTNUM is the desired interrupt level */
    id = InterruptAttachEvent( INTNUM, &event, 0 );

    while (1) {
        InterruptWait( 0, NULL );
        /*
         * do something about the interrupt, 
         * perhaps updating some shared 
         * structures in the resource manager 
         *
         * unmask the interrupt when done
         */
        InterruptUnmask( INTNUM, id );
    }
}

int main(int argc, char **argv) {
    thread_pool_attr_t pool_attr;
    thread_pool_t *tpp;
    dispatch_t *dpp;
    resmgr_attr_t resmgr_attr;
    int id;

    if((dpp = dispatch_create()) == NULL) {
        fprintf( stderr,
            "%s: Unable to allocate dispatch handle.\n", 
            argv[0] );
        return EXIT_FAILURE;
    }

    memset( &pool_attr, 0, sizeof pool_attr );
    pool_attr.handle = dpp;
    pool_attr.context_alloc = dispatch_context_alloc;
    pool_attr.block_func = dispatch_block;
    pool_attr.unblock_func = dispatch_unblock;
```
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

if((tpp = thread_pool_create( &pool_attr,
    POOL_FLAG_EXIT_SELF)) == NULL ) {
    fprintf(stderr,
        "\%s: Unable to initialize thread pool.\n",
        argv[0]);
    return EXIT_FAILURE;
}

iofunc_func_init(_RESMGR_CONNECT_NFUNCS,
    &connect_funcs,
    _RESMGR_IO_NFUNCS, &io_funcs);

iofunc_attr_init( &attr, S_IFNAM | 0666, 0, 0 );
memset( &resmgr_attr, 0, sizeof resmgr_attr );
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

if((id = resmgr_attach( dpp, &resmgr_attr,
    "/dev/mynull",
    _FTYPE_ANY, 0, &connect_funcs,
    &io_funcs,
    &attr )) == -1) {
    fprintf( stderr,
        "\%s: Unable to attach name.\n", argv[0] );
    return EXIT_FAILURE;
}

/* Start the thread which will handle interrupt events. */
pthread_create ( NULL, NULL, interrupt_thread, NULL );

/* Never returns */
thread_pool_start( tpp );

For more examples using the dispatch interface, see
dispatch_create(), message_attach(), and resmgr_attach(). For
information on advanced topics in designing and implementing a
resource manager, see “Combine messages” section of the Writing a
thread_pool_create()

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

dispatch_block(), dispatch_create(), dispatch_unblock(),
pthread_create(), resmgr_attach(), select_attach(),
thread_pool_destroy(), thread_pool_start()
thread_pool_destroy()  
Free the memory allocated to a thread pool

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int thread_pool_destroy( thread_pool_t * pool );
```

Arguments:

- `pool`: A thread pool handle that was returned by `thread_pool_create()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `thread_pool_destroy()` function frees the memory allocated to a thread pool that’s identified by the handle `pool`. This is done only after all the threads of the thread pool have exited.

Prior to QNX Neutrino 6.1.0, this function simply deallocated the thread pool handle and returned. Although this was acceptable for servers that never exited (and consequently never shut down their thread pools), it’s unsuitable for closing down a thread pool.

The `thread_pool_destroy()` function calls the unblock handler provided in the pool attribute structure. The unblock handler is called at least once for every thread in the thread pool. Once the unblock handler is called, the thread calling `thread_pool_destroy()` blocks until the number of threads in the thread pool drops to zero. When there are no more threads in the thread pool, the handle pool is freed and `thread_pool_destroy()` returns.
A side effect of this behavior is that a thread that’s created by the thread pool can’t call `thread_pool_destroy()` because the thread pool count will never drop to zero, and subsequently the function will never return.

**Returns:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred</td>
</tr>
</tbody>
</table>

**Examples:**

```c
#include <sys/dispatch.h>
#include <stdio.h>

int main( int argc, char **argv ) {
    thread_pool_t *tpp;
    ...
    thread_pool_destroy ( tpp );
}
```

For examples using the dispatch interface, see `dispatch_create()`, `message_attach()`, `resmgr_attach()`, and `thread_pool_create()`.

**Classification:**

QNX Neutrino

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

thread_pool_control(), thread_pool_create(), thread_pool_limits(),
thread_pool_start()
thread_pool_limits() © 2007, QNX Software Systems GmbH & Co. KG.
Convenience wrapper function for thread_pool_control()

Synopsis:

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int thread_pool_limits( thread_pool_t * pool,
    int lowater,
    int hiwater,
    int maximum,
    int increment,
    unsigned flags);
```

Arguments:

- **pool** A thread pool handle that was returned by `thread_pool_create()`.
- **lowater** The minimum number of threads that the pool should keep in the blocked state (i.e. threads that are ready to do work), or a negative number if you don’t want to change the current value.
- **hiwater** The maximum number of threads that the pool should keep in the blocked state, or a negative number if you don’t want to change the current value.
- **maximum** The maximum number of threads that the pool can create, or a negative number if you don’t want to change the current value.
- **increment** The number of new threads created at one time, or a negative number if you don’t want to change the current value.
- **flags** The only flag that’s accepted is `THREAD_POOL_CONTROL_NONBLOCK`. For more information, see the documentation for `thread_pool_control()`.
Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The thread_pool_limits() function is a wrapper function for thread_pool_control(). If the value of lowater, hiwater, maximum or increment is ≥0 then that value is adjusted in the thread pool according to the handle pool.

If you don’t set THREAD_POOL_CONTROL_NONBLOCK, the upper and lower bounds for waiting are:

- lower = (lowater != -1) : lowater ? 0;
- upper = (maximum != -1) : maximum ? USHRT_MAX;

Having several threads call this function with the same thread pool handle isn’t recommended.

Returns:

-1 if an error occurs (errno is set).

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
### thread_pool_limits()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `thread_pool_control()`, `thread_pool_create()`,
- `thread_pool_destroy()`, `thread_pool_start()`
thread_pool_start()
Start a thread pool

Synopsis:
#include <sys/dispatch.h>

int thread_pool_start( void *pool );

Arguments:

pool A thread pool handle that was returned by thread_pool_create().

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The thread_pool_start() function starts the thread pool pool. The function may or may not return, depending on the flags that you passed to thread_pool_create().

Returns:

EOK Success.
-1 An error occurred.

Examples:
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
    thread_pool_attr_t pool_attr;
    thread_pool_t *tpp;
    dispatch_t *dpp;
    resmgr_attr_t attr;
    resmgr_context_t *ctp;

    if( (dpp = dispatch_create()) == NULL ) {
thread_pool_start()

```c
    fprintf( stderr, "%s: Unable to allocate \n        dispatch context.\n", argv[0] );
    return EXIT_FAILURE;
}
memset( &pool_attr, 0, sizeof (pool_attr) );
pool_attr.handle = dpp;
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.unblock_func = dispatch_unblock;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

    if( (tpp = thread_pool_create( &pool_attr,
        POOL_FLAG_EXIT_SELF )) == NULL ) {
        fprintf( stderr, "%s: Unable to initialize \n            thread pool.\n", argv[0] );
        return EXIT_FAILURE;
    }

    /* Never returns */
    thread_pool_start( tpp );
```

For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

*continued...*
thread_pool_start()

See also:

thread_pool_create(), thread_pool_destroy()
Synopsis:

```c
#include <sys/neutrino.h>

int ThreadCancel( int tid,
    void (*canstub)(void) );

int ThreadCancel_r( int tid,
    void (*canstub)(void) );
```

Arguments:

- `tid` The ID of the thread that you want to destroy, as returned by `ThreadCreate()`.
- `canstub` A pointer to the location that you want the thread to jump to when the cancellation occurs; see below.

You must provide a `canstub` function.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls request that the thread specified by `tid` be canceled. The target thread’s cancelability state and type determine when the cancellation takes effect.

The `ThreadCancel()` and `ThreadCancel_r()` functions are identical, except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling `pthread_cancel()`.

When the cancellation is acted upon, the thread jumps to the location specified by `cansub`. This stub should call cancellation cleanup handlers for the thread. When the last cancellation cleanup handler returns, the stub must terminate the thread using:

```c
ThreadDestroy( 0, -1, PTHREAD_CANCEL);
```

Unlike `ThreadDestroy()`, which destroys a thread immediately, `ThreadCancel()` requests that the target thread execute any cleanup code and then terminate at its earliest convenience.

The cancellation processing in the target thread runs asynchronously with respect to the calling thread, which doesn’t block.

The combinations of cancelability state and type are as follows:

<table>
<thead>
<tr>
<th>State</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Deferred</td>
<td>Cancel requests are made pending</td>
</tr>
<tr>
<td>Disabled</td>
<td>Async</td>
<td>Cancel requests are made pending</td>
</tr>
</tbody>
</table>
| Enabled | Deferred| Cancellation happens at the next cancellation point. These are at explicitly coded calls to `pthread_testcancel()` or an attempt to enter a blocking state in any of the calls defined in the table below. All kernel calls that block are cancellation points, with the exception of `MsgSendvnc()` and `SyncMutexLock()`.
| Enabled | Async   | Cancellation happens immediately.                                            |
Use `pthread_setcancelstate()` and `pthread_setcanceltype()` to set the state and type.

POSIX defines a list of functions that are cancellation points; some functions that aren’t listed there may also be cancellation points. For a full list, see “Cancellation points” in the appendix, Summary of Safety Information. Any function that calls a blocking kernel call that’s a cancellation point will itself become a cancellation point when the kernel call is made. The most common blocking kernel call in library code is `MsgSendv()`.

### Blocking states

These calls don’t block.

### Returns:

The only difference between these functions is the way they indicate errors:

**ThreadCancel()**

If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.

**ThreadCancel_r()**

EOK is returned on success. This function does **NOT** set `errno`. If an error occurs, any value in the Errors section may be returned.

### Errors:

`ESRCH` The thread indicated by `tid` doesn’t exist.

### Classification:

QNX Neutrino
Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`pthread_cancel()`, `pthread_setcancelstate()`, `pthread_setcanceltype()`, `pthread_testcancel()`, `ThreadCreate()`, `ThreadDestroy()`
ThreadCreate(), ThreadCreate_r()

Synopsis:

```
#include <sys/neutrino.h>

int ThreadCreate(
    pid_t pid,
    void* (func)( void* ),
    void* arg,
    const struct _thread_attr* attr );

int ThreadCreate_r(
    pid_t pid,
    void* (func)( void* ),
    void* arg,
    const struct _thread_attr* attr );
```

Arguments:

pid The ID of the process that you want to create the thread in, or 0 to create the thread in the current process.

func A pointer to the function that you want the thread to execute. The arg argument that you pass to ThreadCreate() is passed to func() as its sole argument. If func() returns, it returns to the address defined in the exitfunc member of attr.

arg A pointer to any data that you want to pass to func.

attr A pointer to a _thread_attr structure that specifies the attributes for the new thread, or NULL if you want to use the default attributes.

If you modify the attributes after creating the thread, the thread isn’t affected.

For more information, see “Thread attributes,” below.
**ThreadCreate(), ThreadCreate_r()**

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

These kernel calls create a new thread of execution, with attributes specified by `attr`, within the process specified by `pid`. If `pid` is zero, the current process is used.

Only the Process Manager can create threads in another process.

The `ThreadCreate()` and `ThreadCreate_r()` functions are identical, except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `pthread_create()`.

The new thread shares all resources of the process in which it’s created. This includes memory, timers, channels and connections. The standard C library contains mutexes to make it thread-safe.

**Thread attributes**

The `_thread_attr` structure pointed to by `attr` contains at least the following members:

- `int flags`  
  See below for a list of `flags`. The default flag is always zero.

- `size_t stacksize`
  The stack size of the thread stack defined in the `stackaddr` member. If `stackaddr` is NULL, then `stacksize` specifies the size of stack to dynamically allocate. If `stacksize` is zero, then 4096 bytes are
ThreadCreate(), ThreadCreate_r()

assumed. The minimum allowed stacksize is defined by PTHREAD_STACK_MIN.

**void* stackaddr**

NULL, or the address of a stack that you want the thread to use. Set the stacksize member to the size of the stack.

If you provide a non-NULL stackaddr, it’s your responsibility to release the stack when the thread dies. If stackaddr is NULL, then the kernel dynamically allocates a stack on thread creation and automatically releases it on the thread’s death.

**void* (exitfunc)(void* status)**

The address to return to if the thread function returns.

The thread returns to exitfunc. This means that the status variable isn’t passed as a normal parameter. Instead, it appears in the return-value position dictated by the CPU’s calling convention (e.g. EAX on an x86, R3 on PPC, V0 on MIPS, and so on).

The exitfunc function normally has to have compiler- and CPU-specific manipulation to access the status data (pulling it from the return register location to a proper local variable). Alternatively, you can write the exitfunc function in assembly language for each CPU.

**int policy**

The scheduling policy, as defined by the SchedSet() kernel call. This member is used only if you set the PTHREAD_EXPLICIT_SCHED flag. If you want the thread to inherit the policy, but you want to specify the scheduling parameters in the param member, set the PTHREAD_EXPLICIT_SCHED flag and set the policy member to SCHED_NOCHANGE.

**struct sched_param param**

A sched_param structure that specifies the scheduling parameters, as defined by the SchedSet()
ThreadCreate(),
ThreadCreate_r()

kernel call. This member is used only if you set the
PTHREAD_EXPLICIT_SCHED flag.

You can set the attr argument’s flags member to a combination of the
following:

PTHREAD_CREATE_JOINABLE (default)
   Put the thread into a zombie state when it terminates. It stays in
   this state until you retrieve its exit status or detach the thread.

PTHREAD_CREATE_DETACHED
   Create the thread in the detached state; it doesn’t become a
   zombie. You can’t call ThreadJoin() for a detached thread.

PTHREAD_INHERIT_SCHED (default)
   Use the scheduling attributes of the creating thread for the new
   thread.

PTHREAD_EXPLICIT_SCHED
   Take the scheduling policy and parameters for the new thread
   from the policy and param members of attr.

PTHREAD_SCOPE_SYSTEM (default)
   Schedule the thread is against all threads in the system.

PTHREAD_SCOPE_PROCESS
   Don’t set this flag; the QNX Neutrino OS implements true
   microkernel threads that have only a system scope.

PTHREAD_MULTISIG_ALLOW (default)
   If the thread dies because of an unblocked, uncaught signal,
   terminate all threads, and hence, the process.

PTHREAD_MULTISIG_DISALLOW
   Terminate only this thread; all other threads in the process are
   unaffected.
PTHREAD_CANCEL_DEFERRED (default)

Cancellation occurs only at cancellation points as defined by ThreadCancel().

PTHREAD_CANCELASYNCHRONOUS

Every opcode executed by the thread is considered a cancellation point. The POSIX and C library aren’t asynchronous-cancel safe.

Signal state

The signal state of the new thread is initialized as follows:

- The signal mask is inherited from the creating thread.
- The set of pending signals is empty.
- The cancel state and type are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED.

Local storage for private data

Each thread contains a thread local storage area for its private data. This area can be accessed using the global variable _TLS defined in <sys/neutrino.h> as a pointer. The kernel ensures that _TLS always points to the thread local storage for the thread that’s running.

The thread local storage is defined by the structure _thread_local_storage, which contains at least the following members:

void* (exitfunc)(void *)

The exit function to call if the thread returns.

void* arg

The sole argument that was passed to the thread.

int* errptr

A pointer to a thread unique errno value. For the main thread, this points to the global variable errno. For all other threads, this points to the member errval in this structure.
ThreadCreate(),
ThreadCreate_r()

**int errval**  
A thread-unique *errno* that the thread uses if it isn’t  
the main thread.

**int flags**  
The thread flags used on thread creation in addition  
to runtime flags used for implementing thread  
cancellation.

**pid_t pid**  
The ID of the process that contains the thread.

**int tid**  
The thread’s ID.

**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate  
errors:

*ThreadCreate()*  
The thread ID of the newly created thread. If an error occurs,  
the function returns -1 and sets *errno*.

*ThreadCreate_r()*

The thread ID of the newly created thread. This function does  
**NOT** set *errno*. If an error occurs, the function returns the  
negative of a value from theErrors section.

**Errors:**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>All kernel thread objects are in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the</td>
</tr>
<tr>
<td></td>
<td>buffers provided.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid scheduling policy or priority specified.</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>PTHREAD_SCOPE_PROCESS was requested. All kernel threads are PTHREAD_SCOPE_SYSTEM.</td>
</tr>
</tbody>
</table>
ThreadCreate(), ThreadCreate_r()

The calling thread doesn’t have sufficient permission to create a thread in another process. Only a thread with a process ID of 1 can create threads in other processes.

ESRCH The process indicated by pid doesn’t exist.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The QNX interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```c
void nothingthread( void )
{
    return;
}
```

See also:

pthread_create(), sched_param, SchedSet(), ThreadCancel(), ThreadDestroy()
Synopsis:

#include <sys/neutrino.h>

int ThreadCtl( int cmd,
    void * data );

int ThreadCtl_r( int cmd,
    void * data );

Arguments:

    cmd  The command you want to execute; see below.
    data A pointer to data associated with the specific command; see below.

Library:

    libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

These kernel calls allow you to make QNX-specific changes to a thread.

The ThreadCtl() and ThreadCtl_r() functions are identical except in the way they indicate errors. See the Returns section for details.

The following calls are defined:

ThreadCtl(_NTO_TCTL_ALIGN_FAULT, data)

    Control the misaligned access response. The data argument is a pointer to an int whose value indicates how you want to respond:

    • Greater than 0 — make a misaligned access fault with a SIGBUS, if the architecture permits it.
ThreadCtl(), ThreadCtl_r()

- Less than 0 — make the kernel attempt to emulate an instruction with a misaligned access. If the attempt fails, it also faults with a SIGBUS.
- 0 — don’t change the alignment-fault handling for the thread.

The function sets data to a positive or negative number, indicating the previous state of the alignment-fault handling.

Threads created by the calling thread inherit the _NTO_TCTL_ALIGN_FAULT status.

ThreadCtl(_NTO_TCTL_NAME, struct _thread_name name)

Set and retrieve the name of the current thread. The data structure contains the following fields:

new_name_len
name_buf_len
name_buf[1]

The name_buf is a continuous buffer that extends the structure that contains space for max(name_buf_len, new_name_len). If new_name_len is

- Less than 0 — a new name isn’t set for the thread.
- 0 — thread name is cleared.
- Greater than 0 — the thread name is set to a new name of name_buf_len bytes starting at name_buf.

The current thread name (or previous name if a new name is being set) is placed as a NULL-terminated string in the buffer pointed to by name_buf. The name returned has a maximum of name_buf_len bytes. No thread name is returned if name_buf_len is zero.
Currently, the thread names are limited to _NTO_THREAD_NAME_MAX.

You can also use the pthread_getname_np() and pthread_getname_np() functions instead of calling ThreadCtl() directly.

This call was defined in the QNX Neutrino Core OS 6.3.2.

ThreadCtl(_NTO_TCTL_IO, 0)

Request I/O privileges; let the thread execute the I/O opcodes in, ins, out, outs, cli, sti on architectures where it has the appropriate privilege, and let it attach IRQ handlers.

You need root permissions to use this command. If a thread attempts to use these opcodes without successfully executing this call, the thread faults with a SIGSEGV when the opcode is attempted.

Normally, user threads can’t read or write to memory used by the procnto microkernel. It’s, however, too expensive to maintain this protection for memory on a PPC8xx family processor (that runs procnto-800), once a thread has been called with ThreadCtl(_NTO_TCTL_IO, 0) operation. This means that device driver threads on a PPC8xx family processor are available to read, and more importantly, write to memory being used by the kernel and the process manager. Since this causes system instability, you should take appropriate care not to damage the memory.

Threads created by the calling thread inherit the _NTO_TCTL_IO status.

ThreadCtl(_NTO_TCTL_ONE_THREAD_HOLD, (void *) tid)

Hold one thread, with tid as thread ID, in the process. Returns an error ESRCH when tid is not the existing thread.
**ThreadCtl(), ThreadCtl_r()** © 2007, QNX Software Systems GmbH & Co. KG.

This call was defined in the QNX Neutrino Core OS 6.3.2.

ThreadCtl(_NTO_TCTL_ONE_THREAD_CONT, (void *) tid)

Continue one thread, with tid as thread ID, in the process.

Returns an error ESRCH when tid is not the existing thread.

This call was defined in the QNX Neutrino Core OS 6.3.2.

ThreadCtl(_NTO_TCTL_RUNMASK, (int)runmask)

Set processor affinity for the calling thread in a multiprocessor system. Each set bit in runmask represents a processor that the thread can run on.

By default, a thread’s runmask is set to all ones, which allows it to run on any available processor. A value of 0x01 would, for example, force the thread to only run on the first processor.

You can use _NTO_TCTL_RUNMASK to optimize the runtime performance of your system by, for example, relegating nonrealtime threads to a specific processor. In general, this shouldn’t be necessary, since the QNX realtime scheduler always preempts a lower-priority thread immediately when a higher priority thread becomes ready.

The main effect of processor locking is the effectiveness of the CPU cache, since threads can be prevented from migrating.

Threads created by the calling thread don’t inherit the specified runmask.

ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET, (int *) data)

Get and set the runmask (the processor affinity) to a proper value for the calling thread in a multiprocessor system. The data parameter is a pointer to an integer. On input, the pointer to value is used to set the new runmask for the thread (see _NTO_TCTL_SET_RUNMASK for details.) After the function has completed, the contents of *data will be replaced with the
previous runmask for the thread. Calling ThreadCtl again with the same pointer will restore the runmask to the state before the call.

ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT,
&(struct_thread_runmask *) data)
Manipulate calling thread’s runmask and inherit mask. This call takes a pointer to a struct_thread_runmask. The values for both masks at the time of the call are returned in their respective members of the passed in struct. If the masks are passed as zero at the time of the call, the masks are left unaltered, otherwise they are set to the specified value(s). See example below.

This call was defined in the QNX Neutrino Core OS 6.3.2.

ThreadCtl(_NTO_TCTL_THREADS_CONT, 0)
Unfreeze all threads in the current process that were frozen using the _NTO_TCTL_THREADS_HOLD command.

ThreadCtl(_NTO_TCTL_THREADS_HOLD, 0)
Freeze all threads in the current process except the calling thread.

Threads created by the calling thread aren’t frozen.

Blocking states
These calls don’t block.

Returns:
The only difference between these functions is the way they indicate errors:

ThreadCtl() If an error occurs, the function returns -1 and sets errno. Any other value returned indicates success.
ThreadCtl(), ThreadCtl_r()

ThreadCtl_r() EOK is returned on success. This function does NOT set errno. If an error occurs, any value in the Errors section may be returned.

Errors:

E2BIG The name is larger than the accepted size.
EINV AL The name buffer length is invalid or smaller that the new name length.
EPERM The process doesn’t have superuser capabilities.
ESRCH The tid is not the existing thread.

Examples:

```c
#include <sys/neutrino.h>
#include <sys/syspage.h>
#include <malloc.h>
#include <stdio.h>

int main(void)
{
    int *rsizep, rsize, size_tot;
    unsigned *rmaskp, *inheritp;
    unsigned buf[8];
    void *freep;

    /*
    * struct _thread_runmask is not
    * uniquely sized so we construct
    * our own.
    */

    rsize = RMSK_SIZE(_syspage_ptr->num_cpu);

    size_tot = sizeof(*rsizep);
    size_tot += sizeof(*rmaskp) * rsize;
    size_tot += sizeof(*inheritp) * rsize;

    return 0;
}
```
if (size_tot <= sizeof(buf)) {
    rsizep = buf;
    freep = NULL;
} else if ((rsizep = freep = malloc(size_tot)) == NULL) {
    perror("malloc");
    return 1;
}

memset(rsizep, 0x00, size_tot);

*rsizep = rsize;
rmaskp = (unsigned *)(rsizep + 1);
inheritp = rmaskp + rsize;

/*
 * Both masks set to 0 means get current
 * values without alteration.
 */
if (ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT, rsizep) == -1)
    perror("_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT");
free(freep);
return 1;

/*
 * Restrict our inherit mask to the last cpu, leave
 * runmask unaltered.
 */
memset(rsizep, 0x00, size_tot);
*rsizep = rsize;
RMSK_SET(_syspage_ptr->num_cpu - 1, inheritp);
if (ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT, rsizep) == -1)
    perror("_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT");
free(freep);
return 1;

free(freep);
ThreadCtl(), ThreadCtl_r()

```c
return 0;
```

### Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

### See also:

InterruptDisable(), InterruptEnable(), InterruptMask(), InterruptUnmask()
Synopsis:

```c
#include <sys/neutrino.h>

int ThreadDestroy( int tid,
                   int priority,
                   void* status );

int ThreadDestroy_r( int tid,
                     int priority,
                     void* status );
```

Arguments:

- **tid**: The ID of the thread that you want to destroy, as returned by `ThreadCreate()`, or 0 to destroy the current thread, or -1 to destroy all the threads in the current process.
- **priority**: The priority at which you want to destroy multiple threads, or -1 to use the priority of the current thread.
- **status**: The value to make available to a to a thread that joins a nondetached thread that’s destroyed.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls terminate the thread specified by `tid`. If `tid` is 0, the calling thread is assumed. If `tid` is -1, all of the threads in the process are destroyed. When multiple threads are destroyed, the destruction is scheduled one thread at a time at the priority specified by the `priority` argument. If `priority` is -1, then the priority of the calling thread is used.
The `ThreadDestroy()` and `ThreadDestroy_r()` functions are identical, except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `pthread_abort()` or `pthread_exit()`

If a terminated thread isn’t detached, it makes the value specified by the `status` argument available to any successful join on it. Until another thread retrieves this value, the thread ID `tid` isn’t reused, and a small kernel resource (a thread object) is held in the system. If the thread is detached, then `status` is ignored, and all thread resources are immediately released.

When the last thread in a process is destroyed, the process terminates, and all thread resources are released, even if they’re not detached and unjoined.

On return from `ThreadDestroy()` or `ThreadDestroy_r()`, the target thread is marked for death, but if it isn’t possible to kill it immediately, it may not be terminated until it attempts to run.

**Blocking states**

If these calls return, they don’t block.

**Returns:**

If the calling thread is destroyed, `ThreadDestroy()` and `ThreadDestroy_r()` don’t return.

The only difference between these functions is the way they indicate errors:

`ThreadDestroy()`

If this function returns and an error occurs, -1 is returned and `errno` is set. Any other value returned indicates success.
ThreadDestroy_r()

EOK is returned on success. This function does **NOT** set `errno`. If this function returns and an error occurs, any value in the Errors section may be returned.

**Errors:**

- ESRCH The thread indicated by `tid` doesn’t exist.

**Classification:**

- QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

- `pthread_abort()`, `pthread_exit()`, `ThreadCancel()`, `ThreadCreate()`
**ThreadDetach(), ThreadDetach_r()** © 2007, QNX Software Systems GmbH & Co. KG.

**Detach a thread from a process**

**Synopsis:**

```c
#include <sys/neutrino.h>

int ThreadDetach( int tid );

int ThreadDetach_r( int tid );
```

**Arguments:**

- `tid` The ID of the thread that you want to detach, as returned by `ThreadCreate()`, or 0 to detach the current thread.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

These kernel calls detach the thread specified by `tid`. If `tid` is zero, the calling thread is used. Once detached, attempts to call `ThreadJoin()` on `tid` fail. When a detached thread terminates, its termination status is discarded and all its resources are released.

The `ThreadDetach()` and `ThreadDetach_r()` functions are identical, except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `pthread_detach()`.

**Blocking states**

These calls don’t block.
ThreadDetach(), ThreadDetach_r()

Returns:

The only difference between these functions is the way they indicate errors:

ThreadDetach()  
If an error occurs, the function returns -1 and sets errno. Any other value returned indicates success.

ThreadDetach_r()  
Returns EOK on success. This function does NOT set errno. If an error occurs, the function can return any value listed in the Errors section.

Errors:

EINVAL    The thread is already detached.
ESRCH     The thread indicated by tid doesn’t exist.

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Interrupt handler</th>
<th>Signal handler</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_detach(), ThreadCreate(), ThreadJoin()
ThreadJoin(), ThreadJoin_r()

Block until a thread terminates

Synopsis:

```
#include <sys/neutrino.h>

int ThreadJoin( int tid,
                void** status );

int ThreadJoin_r( int tid,
                  void** status );
```

Arguments:

- **tid**  
  The ID of the thread that you want to detach, as returned by `ThreadCreate()`.

- **status**  
  The address of a pointer to a location where the function can store the thread’s exit status.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ThreadJoin()` and `ThreadJoin_r()` kernel calls block until the thread specified by `tid` terminates. If `status` isn’t NULL, the functions save the thread’s exit status in the area pointed to by `status`. If the thread `tid` has already terminated, the functions immediately return with success and the status, if requested.

These functions are identical except in the way they indicate errors. See the Returns section for details.
Instead of using these kernel calls directly, consider calling `pthread_join()` or `pthread_timedjoin()`.

When `ThreadJoin()` returns successfully, the target thread has been successfully terminated. Until this occurs, the thread ID `tid` isn’t reused and a small kernel resource (a thread object) is retained.

You can’t join a thread that’s detached (see `ThreadCreate()` and `ThreadDetach()`).

The target thread must be joinable. Multiple `pthread_join()`, `pthread_timedjoin()`, `ThreadJoin()`, and `ThreadJoin_r()` calls on the same target thread aren’t allowed.

**Blocking states**

- **STATE_JOIN** The calling thread blocks waiting for the indicated thread to exit.

**Returns:**

The only difference between these functions is the way they indicate errors:

- `ThreadJoin()` If an error occurs, the function returns -1 and sets `errno`. Any other value returned indicates success.
- `ThreadJoin_r()` Returns EOK on success. This function does **NOT** set `errno`. If an error occurs, the function may return any value listed in the Errors section.

**Errors:**

- **EBUSY** Attempt to join a thread which has been joined by another thread.
- **EDEADLK** Attempt to join to yourself.
- **EFAULT** A fault occurred when the kernel tried to access `status`.

September 10, 2007
ThreadJoin(), ThreadJoin_r()

EINTR  The call was interrupted by a signal.
EINVAL  Attempt to join a thread which is detached (see ThreadDetach()).
ESRCH  The thread indicated by tid doesn’t exist.
ETIMEDOUT  A kernel timeout unblocked the call. See TimerTimeout().

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_join(), pthread_timedjoin(), ThreadCreate(), ThreadDetach()
time()

Determine the current calendar time

Synopsis:

```c
#include <time.h>

time_t time( time_t * tloc );
```

Arguments:

- `tloc` NULL, or a pointer to a `time_t` object where the function can store the current calendar time.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `time()` function takes a pointer to `time_t` as an argument and returns a value of `time_t` on exit. The returned value is the current calendar time, in seconds, since the Unix Epoch, 00:00:00 January 1, 1970 Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

You typically use the `date` command to set the computer’s internal clock using Coordinated Universal Time (UTC). Use the `TZ` environment variable or `_CS_TIMEZONE` configuration string to establish the local time zone. For more information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.

Returns:

The current calendar time, in seconds, since 00:00:00 January 1, 1970 Coordinated Universal Time (UTC). If `tloc` isn’t NULL, the current calendar time is also stored in the object pointed to by `tloc`. 
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main( void )
{
    time_t time_of_day;
    time_of_day = time( NULL );
    printf( "It is now: %s", ctime( &time_of_day ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
It is now: Wed Jun 30 09:09:33 1999
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`asctime`, `asctime_r`, `clock`, `clock_gettime`, `ctime`, `difftime`, `gmtime`, `localtime`, `localtime_r`, `mktime`, `strftime`, `tzset`

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User's Guide
timer_create()
Create a timer

Synopsis:

```c
#include <signal.h>
#include <time.h>

int timer_create( clockid_t clock_id,
                  struct sigevent * evp,
                  timer_t * timerid );
```

Arguments:

- `clock_id` The clock source that you want to use; one of:
  - `CLOCK_REALTIME` — the standard POSIX-defined timer.
  - `CLOCK_SOFTTIME` — currently, the same as `CLOCK_REALTIME`.

- `evp` NULL, or a pointer to a `sigevent` structure containing the event that you want to deliver when the timer fires.

- `timerid` A pointer to a `timer_t` object where the function stores the ID of the new timer.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_create()` function creates a per-process timer using the specified clock source, `clock_id`, as the timing base.

You can use the time ID that the function stores in `timerid` in subsequent calls to `timer_gettime()`, `timer_settime()`, and `timer_delete()`.

The timer is created in the disabled state, and isn’t enabled until you call `timer_settime()`.
We recommend the following event types:

- SIGEV_SIGNAL
- SIGEV.Signal.CODE
- SIGEV.Signal.THREAD
- SIGEV_PULSE

If the `evp` argument is NULL, a SIGALRM signal is sent to your process when the timer expires. To specify a handler for this signal, call `sigaction()`.

**Returns:**

0 Success. The `timerid` argument is set to the timer’s ID.

-1 An error occurred (`errno` is set).

**Errors:**

EAGAIN All timers are in use. You’ll have to wait for a process to release one.

EINVAL The `clock_id` isn’t one of the valid `CLOCK_` constants.

**Examples:**

```c
/*
 * Demonstrate how to set up a timer that, on expiry,
 * sends us a pulse. This example sets the first
 * expiry to 1.5 seconds and the repetition interval
 * to 1.5 seconds.
 */

#include <stdio.h>
#include <time.h>
#include <sys/netmgr.h>
#include <sys/neutrino.h>

#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL

typedef union {
    struct _pulse pulse;
```
main()
{
    struct sigevent event;
    struct itimerspec itime;
    timer_t timer_id;
    int chid;
    int rcvid;
    my_message_t msg;

    chid = ChannelCreate(0);

    event.sigev_notify = SIGEV_PULSE;
    event.sigev_coid = ConnectAttach(ND_LOCAL_NODE, 0,
                        _NTO_SIDE_CHANNEL, 0);
    event.sigev_priority = getprio(0);
    event.sigev_code = MY_PULSE_CODE;
    timer_create(CLOCK_REALTIME, &event, &timer_id);

    itime.it_value.tv_sec = 1;
    /* 500 million nsecs = .5 secs */
    itime.it_value.tv_nsec = 500000000;
    itime.it_interval.tv_sec = 1;
    /* 500 million nsecs = .5 secs */
    itime.it_interval.tv_nsec = 500000000;
    timer_settime(timer_id, 0, &itime, NULL);

    /*
     * As of the timer_settime(), we will receive our pulse
     * in 1.5 seconds (the itime.it_value) and every 1.5
     * seconds thereafter (the itime.it_interval)
     */

    for (;;) {
        rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
        if (rcvid == 0) { /* we got a pulse */
            if (msg.pulse.code == MY_PULSE_CODE) {
                printf("we got a pulse from our timer\n");
            } /* else other pulses ... */
        } /* else other messages ... */
    }
}
Classification:

POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

The QNX Neutrino version of `timer_create()` is different from the QNX 4 version, which was based on a draft standard.

See also:

`clock_getres()`, `clock_gettime()`, `clock_settime()`, `nanosleep()`,
`_pulse`, `sigaction()`, `sigevent`, `sleep()`, `TimerCreate()`,
`timer_delete()`, `timer_getexpstatus()`, `timer_getoverrun()`,
`timer_gettime()`, `timer_settime()`
timer_delete()  

Delete a timer

Synopsis:

```c
#include <time.h>

int timer_delete( timer_t timerid );
```

Arguments:

- `timerid`  
  A `timer_t` object that holds a timer ID, as set by `timer_create()`.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_delete()` function removes a previously attached timer based upon the `timerid` returned from the `timer_create()` function. The timer is removed from the active system timer list, and returned to the free list of available timers.

Returns:

- 0  
  Success.

-1  
  An error occurred (`errno` is set).

Errors:

- `EINVAL`  
  The timer `timerid` isn’t attached to the calling process.

Classification:

POSIX 1003.1 TMR
timer_delete()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

clock_getres(), clock_gettime(), clock_settime(), nanosleep(), sleep(),
timer_create(), timer_getexpstatus(), timer_getoverrun(),
timer_gettime(), timer_settime()
timer_getexpstatus()  
Get the expiry status of a timer

Synopsis:

```
#include <time.h>

int timer_getexpstatus( timer_t timerid );
```

Arguments:

`timerid`  
A `timer_t` object that holds a timer ID, as set by `timer_create()`. 

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `timer_getexpstatus()` function gets the expiry status of the time with the ID given by `timerid`.

This function is based on a POSIX draft. If you want your code to be more portable, then instead of calling `timer_getexpstatus()`, call `timer_gettime()` and check the amount of time left on the timer.

Returns:

0   The timer has expired.

-1   An error occurred (`errno` is set).

Errors:

EINVAL   The timer specified by `timerid` doesn’t exist.
Classification:

QNX Neutrino

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

timer_create(), timer_delete(), timer_getoverrun(), timer_gettime(),
timer_settime(), TimerInfo()
Synopsis:

```c
#include <signal.h>
#include <time.h>

int timer_getoverrun( timer_t timerid );
```

Arguments:

- `timerid` A `timer_t` object that holds a timer ID, as set by `timer_create()`.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

When a timer expiration signal is received by a process, the `timer_getoverrun()` function returns the timer expiration overrun count for the timer specified by `timerid`.

Only a single signal is queued to the process for a given timer at any point in time. When a timer that has a signal pending expires, no signal is queued and a timer overrun occurs.

The overrun count returned is the number of extra timer expirations that occurred between the time the expiration signal was queued and when it was delivered or accepted, up to but not including `DELAYTIMER_MAX`. If the number of overruns is greater than or equal to `DELAYTIMER_MAX`, the overrun count is set to `DELAYTIMER_MAX`.

The value returned by `timer_getoverrun()` applies to the most recent expiration signal for the specified timer. If no expiration signal has been delivered, the overrun count is 0.
Returns:
The number of overruns, or -1 if an error occurs (errno is set).

Errors:
EINV VAL Invalid timer timerid.

Classification:
POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:
timer_create(), timer_delete(), timer_getexpstatus(), timer_gettime(),
timer_settime(), TimerInfo()
timer_gettime()  
Get the amount of time left on a timer

Synopsis:
```c
#include <time.h>

int timer_gettime( timer_t timerid,
                   struct itimerspec *value );
```

Arguments:
- `timerid` A `timer_t` object that holds a timer ID, as set by `timer_create()`.
- `value` A pointer to a `itimerspec` structure that the function fills in with the timer’s time until expiry. The structure contains at least the following members:

  ```c
  struct timespec it_value
  
  A `timespec` structure that contains the amount of time left before the timer expires, or zero if the timer is disarmed. This value is expressed as the relative interval until expiration, even if the timer was armed with an absolute time.
  
  struct timespec it_interval
  
  A `timespec` structure that contains the timer’s reload value. If nonzero, it indicates a repetitive timer period.
  ```

Library:
- `libc`

Use the `-l libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `timer_gettime()` function gets the amount of time left before the specified timer is to expire, along with the timer’s reload value, and stores it in the space provided by the `value` argument.
timer_gettime() © 2007, QNX Software Systems GmbH & Co. KG.

Returns:

0  Success.
-1  An error occurred (errno is set).

Errors:

EINVAL  The timer timerid isn’t attached to the calling process.

Classification:

POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

clock_getres(), clock_gettime(), clock_settime(), nanosleep(), sleep(),
timer_create(), timer_delete(), timer_getexpstatus(),
timer_getoverrun(), timer_settime(), timespec
timer_settime()

Set the expiration time for a timer

Synopsis:

#include <time.h>

int timer_settime( timer_t timerid,  
    int flags,  
    struct itimerspec * value,  
    struct itimerspec * ovalue );

Arguments:

timerid A timer_t object that holds a timer ID, as set by timer_create().

flags The type of timer to arm if you aren’t disarming the timer. The valid bits include:

- TIMER_ABSTIME — the it_value represents an absolute expiration date in seconds and nanoseconds from 1970. If the date specified has already passed, the function succeeds and the expiration notice is made. If you don’t set this bit, the it_value represents a relative expiration period that’s offset from the current system time by the specified number of seconds and nanoseconds.

value A pointer to a itimerspec structure that specifies the value that you want to set for the timer’s time until expiry. For more information, see timer_gettime().

ovalue NULL, or a pointer to a itimerspec structure that the function fills in with the timer’s former time until expiry.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.
Description:

The `timer_settime()` function sets the expiration time of the timer specified by `timerid` from the `it_value` member of the `value` argument. If the `it_value` structure member of `value` is zero, then the timer is disarmed.

If the `it_interval` member of `value` is nonzero, then it specifies a repeat rate that is added to the timer once the `it_value` period has expired. Subsequently, the timer is automatically rearmed, causing it to become continuous with a period of `it_interval`.

If the `ovalue` parameter isn’t NULL, then on return from this function it contains a value representing the previous amount of time left before the timer was to have expired, or zero if the timer was disarmed. The previous interval timer period is also stored in the `it_interval` member.

The `timerid` is local to the calling process, and must have been created using `timer_create()`.

Returns:

0  Success.

-1  An error occurred (`errno` is set).

Errors:

EFAULT  A fault occurred trying to access the buffers provided.

EINVAL  The timer `timerid` isn’t attached to the calling process or the number of nanoseconds specified by the `tv_nsec` member of one of the `timespec` structures in the `itimerspec` structure pointed to by `value` is less than zero or greater than or equal to 1000 million.
Examples:

See `timer_create()`.

Classification:

POSIX 1003.1 TMR

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`clock_getres()`, `clock_gettime()`, `clock_settime()`, `errno`, `nanosleep()`,
`sleep()`, `timer_create()`, `timer_delete()`, `timer_getexpstatus()`,
`timer_getoverrun()`, `timer_gettime()`
Set a timeout on a blocking state

Synopsis:

```c
#include <time.h>

extern int timer_timeout(
    clockid_t id,
    int flags,
    const struct sigevent* notify,
    const struct timespec* ntime,
    struct timespec* otime);

extern int timer_timeout_r(
    clockid_t id,
    int flags,
    const struct sigevent* notify,
    const struct timespec* ntime,
    struct timespec* otime);
```

Arguments:

* id  The type of timer used to implement the timeout. The possible clock types of *id* are:

  CLOCK_MONOTONIC

  A clock that always increases at a constant rate and can’t be adjusted.

  CLOCK_SOFTTIME

  Same as CLOCK_REALTIME, but if the CPU is in powerdown mode, the clock stops running.

  CLOCK_REALTIME

  A clock that maintains the system time.

* flags  A bitmask that specifies which states you want the timeout to apply to; see below.

* notify  A pointer to a *sigevent* structure that defines the event that the kernel acts on if the timeout expires; see below.

* ntime  A pointer to a *timespec* structure that specifies the timeout.
otine NULL, or a pointer to a location where the function can store the time remaining in the sleep.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The timer_timeout() and timer_timeout_r() functions are identical except in the way they indicate errors. See the Returns section for details.

The timer_timeout() function sets the timeout ntime on any kernel blocking state. The actual timeout that occurred is returned in otime. The resolution of otime for both timer_timeout() and TimerTimeout() functions is in nanoseconds. The difference, however, for otime in these two functions is in the format. For timer_timeout(), the otime is a pointer to timespec structure with two integers, whereas for TimerTimeout(), the pointer is of uint64_t type.

The time in TimerTimeout()’s ntime argument is also in nanoseconds. When ntime is passed to TimerTimeout(), the time (in timespec) is converted from seconds and nanoseconds into nanoseconds. When otime is returned to timer_timeout(), the time is converted from nanoseconds into seconds and nanoseconds.

The kernel blocking states are entered as a result of the following kernel calls:

<table>
<thead>
<tr>
<th>Kernel function call</th>
<th>Blocking state</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterruptWait()</td>
<td>STATE_INTR</td>
</tr>
<tr>
<td>MsgReceivev()</td>
<td>STATE_RECEIVE</td>
</tr>
</tbody>
</table>

continued…
The user specifies which states the timeout should apply to via a bitmask passed in the *flags* argument. The bits are defined by the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_TIMEOUT_CONDVAR</td>
<td>Timeout on STATE_CONDVAR.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_JOIN</td>
<td>Timeout on STATE_JOIN.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_INTR</td>
<td>Timeout on STATE_INTR.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_MUTEX</td>
<td>Timeout on STATE_MUTEX.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_RECEIVE</td>
<td>Timeout on STATE_RECEIVE.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_REPLY</td>
<td>Timeout on STATE_REPLY.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_SEM</td>
<td>Timeout on STATE_SEM.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_SEND</td>
<td>Timeout on STATE_SEND.</td>
</tr>
<tr>
<td>_NTO_TIMEOUT_SIGSUSPEND</td>
<td>Timeout on STATE_SIGSUSPEND.</td>
</tr>
</tbody>
</table>

Continued...
timer_timeout(),
timer_timeout_r()

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_TIMEOUT_SIGWAITINFO</td>
<td>Timeout on STATE_SIGWAITINFO.</td>
</tr>
</tbody>
</table>

For example, to set a timeout on `MsgSendv()`, specify:

\[
\text{\_NTO\_TIMEOUT\_SEND} \mid \text{\_NTO\_TIMEOUT\_REPLY}
\]

Once a timeout is specified using `timer_timeout()`, it’s armed and released under the following conditions:

- **Armed**: The kernel attempts to enter a blocking state specified in `flags`.
- **Released**: One of the above kernel calls completed without blocking, or the kernel call blocks but unblocks before the timeout expires, or the timeout expires.

The `timer_timeout()` function always operates on a one-shot basis. When one of the above kernel calls returns (or is interrupted by a signal), the timeout request is removed from the system. Only one timeout per thread may be in effect at a time. A second call to `timer_timeout()`, without calling one of the above kernel functions, replaces the existing timeout on that thread. A call with `flags` set to zero ensures that a timeout won’t occur on any state. This is the default when a thread is created.

Always call `timer_timeout()` just before the function that you wish to timeout. For example:

```c
...
event.sigev_notify = SIGEV_UNBLOCK;

timeout.tv_sec = 10;
timeout.tv_nsec = 0;

timer_timeout( CLOCK_REALTIME,
               \text{\_NTO\_TIMEOUT\_SEND} \mid \text{\_NTO\_TIMEOUT\_REPLY},
               &event, &timeout, NULL );

MsgSendv( coid, NULL, 0, NULL, 0 );
...```

September 10, 2007
If the signal handler is called between the calls to `timer_timeout()` and `MsgSendv()`, the `timer_timeout()` values are saved during the signal handler and then are restored when the signal handler exits.

If the timeout expires, the kernel acts upon the event specified by the `sigevent` structure pointed to by the `notify` argument. We recommend the following event types in this case:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE
- SIGEV_UNBLOCK
- SIGEV_INTR

Only SIGEV_UNBLOCK guarantees that the kernel call unblocks. A signal may be ignored, blocked, or accepted by another thread and a pulse can only unblock a `MsgReceivev()`. If a NULL is passed for `event` then SIGEV_UNBLOCK is assumed. In this case, a timed out kernel call will return failure with an error of ETIMEDOUT.

`MsgSendv()` won’t unblock on SIGEV_UNBLOCK if the server has already received the message via `MsgReceivev() and` has specified _NTO_CHF_UNBLOCK in the flags argument to its `ChannelCreate()` call. In this case, it’s up to the server to do a `MsgReplyv()` or `MsgError()`.

The timeout:

- Is specified by the `ntime` argument.
- Is relative to the current time (when `timer_timeout()` is called), unless `flags` includes TIMER_ABSTIME, which makes the timeout occur at the absolute time set in `ntime`.
- Occurs on a clock tick (see `ClockPeriod()`) so the actual wakeup time is a minimum of:


\[
    (tv_{sec} \times 1000000000 + tv_{nsec}) \div \text{size of timer tick}\text{ nanoseconds}
\]

where \(tv_{sec}\) and \(tv_{nsec}\) are fields of the \textit{timespec} structure (defined in \textit{<time.h>}).

If you specify a resolution that amounts to 1.7 timer ticks, you may wakeup anywhere from 1 to 1.99999999... timer ticks.

If you don’t wish to block for any time, you can pass a NULL for \(ntime\) in which case no timer is used, the event is assumed to be SIGEV_UNBLOCK and an attempt to enter a blocking state as set by \textit{flags} will immediately return with ETIMEDOUT. Although a questionable practice, this can be used to poll potential blocking kernel calls. For example, you can poll for messages using \textit{MsgReceivev()} with an immediate timeout. A much better approach is to use multiple threads and have one block waiting for messages.

If \textit{flags} is set to \_NTO\_TIMEOUT\_NANOSLEEP, then these calls block in the \textit{STATE\_NANOSLEEP} state until the timeout (or a signal which unblocks the thread) occurs. This can be used to implement an efficient kernel sleep as follows:

\[
    \text{timer\_timeout}(\text{CLOCK\_REALTIME, } \_\text{NTO\_TIMEOUT\_NANOSLEEP,} \nonumber \text{ NULL, } \&ntime, \&otime);
\]

If \textit{otime} isn’t NULL and the sleep is unblocked by a signal then it contains the time remaining in the sleep.

**Blocking states**

The kernel calls don’t block unless \_NTO\_TIMEOUT\_NANOSLEEP is specified in \textit{flags}. In this case, the calls block as follows:

\[
    \text{STATE\_NANOSLEEP}
\]

The calling thread blocks for the requested time period.

**Returns:**

\textit{timer\_timeout()}

The previous flags. If an error occurs, the function returns -1 and sets \textit{errno}. 
timer_timeout(), timer_timeout_r()

The previous flags. This function does NOT set errno. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

- **EAGAIN**: All kernel timer entries are in use.
- **EFAULT**: A fault occurred when the kernel tried to access ntime, otime, or notify.
- **EINTR**: The call was interrupted by a signal.
- **EINVAL**: The clock type id isn’t one of CLOCK_MONOTONIC, CLOCK_SoftTime, or CLOCK_REALTIME.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

The timeout value starts timing out when timer_timeout() is called, not when the blocking state is entered. It might be possible to get preempted after calling timer_timeout() but before the blocking kernel call.
See also:

sigevent, TimerCreate(), TimerInfo(), TimerTimeout()
Send an alarm signal

Synopsis:

```c
#include <sys/neutrino.h>

int TimerAlarm( clockid_t id,  
                const struct _itimer * itime,  
                struct _itimer * otime );

int TimerAlarm_r( clockid_t id,  
                 const struct _itimer * itime,  
                 struct _itimer * otime );
```

Arguments:

id  The timer type to use to implement the alarm; one of:

- CLOCK_REALTIME — This is the standard POSIX-defined clock. Timers based on this clock should will wake up the processor if it’s in a power-saving mode.

- CLOCK_SOFTTIME — This clock is only active when the processor is not in a power-saving mode. For example, an application using a CLOCK_SOFTTIME timer to sleep wouldn’t wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode. While the processor isn’t in a power-saving mode, CLOCK_SOFTTIME behaves the same as CLOCK_REALTIME.

- CLOCK_MONOTONIC — This clock always increases at a constant rate and can’t be adjusted.

itime NULL, or a pointer to a _itimer structure that specifies the length of time to wait.

otime NULL, or a pointer to a _itimer structure where the function can store the old timer trigger time.
TimerAlarm(),
TimerAlarm_r()

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls set an alarm signal (SIGALRM) to be delivered to the thread waiting on the timer at the time specified by `itime`. If `otime` isn’t NULL, the old timer trigger time is returned.

The `TimerAlarm()` and `TimerAlarm_r()` functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `alarm()` or `setitimer()`.

Alarm requests aren’t stacked; only a single SIGALRM may be outstanding on a timer at one time. If you call `TimerAlarm()` while an alarm is outstanding, the alarm is reset to the new value passed in `itime`.

If `itime` is NULL, any previous alarm request is canceled, and no new alarm is set.

Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

```
TimerAlarm()  If an error occurs, -1 is returned and `errno` is set.
Any other value returned indicates success.
```

```
KG.

TimerAlarm_r()  EOK is returned on success. This function does
NOT set errno. If an error occurs, any value in the
Errors section may be returned.

Errors:
EAGAIN  All kernel timer entries are in use.
EINVAL  Invalid timer value id.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:
The alarm(), TimerAlarm(), and ualarm() requests aren’t stacked;
only a single SIGALRM generator can be scheduled with these
functions. If the SIGALRM signal hasn’t been generated, the next call
to alarm(), TimerAlarm(), or ualarm() reschedules it.

See also:
alarm(), setitimer(), TimerCreate(), ualarm()
TimerCreate(),

TimerCreate_r()

Create a timer for a process

Synopsis:

```c
#include <sys/neutrino.h>

int TimerCreate( clockid_t id,
                 const struct sigevent *event );

int TimerCreate_r( clockid_t id,
                   const struct sigevent *event );
```

Arguments:

- **id**  
The timing base; supported types are:
  - CLOCK_REALTIME — This is the standard POSIX-defined clock. Timers based on this clock should will wake up the processor if it's in a power-saving mode.
  - CLOCK_SOFTTIME — This clock is only active when the processor is not in a power-saving mode. For example, an application using a CLOCK_SOFTTIME timer to sleep wouldn't wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode. While the processor isn't in a power-saving mode, CLOCK_SOFTTIME behaves the same as CLOCK_REALTIME.
  - CLOCK_MONOTONIC — This clock always increases at a constant rate and can't be adjusted.

- **event**  
NULL, or a pointer to a sigevent structure that contains the event to deliver when the timer fires; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The TimerCreate() and TimerCreate_r() kernel calls create a per-process timer using the clock specified by id as the timing base. These functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling timer_create().

Use the returned timer ID in subsequent calls to the other timer functions.

The timer is created in the disabled state, and isn’t enabled until you call TimerSettime().

The sigevent structure pointed to by event contains the event to deliver when the timer fires. We recommend the following event types in this case:

- If your process executes in a loop using MsgReceivev(), then SIGEV_PULSE is a convenient way of receiving timer pulses.

- If you use signals for event notification, note that signals are always delivered to the process and not directly to the thread that created or armed the timer. You can change this by using a sigev_notify of SIGEV_SIGNAL THREAD.

- The notify types of SIGEV_UNBLOCK and SIGEV_INTR, while allowed, are of questionable use with timers. SIGEV_UNBLOCK is typically used by the TimerTimeout() kernel call, and SIGEV_INTR is typically used with the InterruptWait() kernel call.

If the event argument is NULL, a SIGALRM signal is sent to your process when the timer expires. To specify a handler for this signal, call sigaction().
**TimerCreate(), TimerCreate_r()**

**Blocking states**

These calls don’t block.

**Returns:**

The only difference between these functions is the way they indicate errors:

*TimerCreate()*    The timer ID of the newly created timer. If an error occurs, -1 is returned and *errno* is set.

*TimerCreate_r()*    The timer ID of the newly created timer. This function does **NOT** set *errno*. If an error occurs, the negative of a value from the Errors section is returned.

**Errors:**

- **EINVAL**    The clock ID isn’t valid.
- **EAGAIN**    All kernel timer objects are in use.
- **EFAULT**    A fault occurred when the kernel tried to access the buffers provided.

**Classification:**

QNX Neutrino

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`sigevent`, `timer_create()`, `TimerAlarm()`, `TimerDestroy()`, `TimerInfo()`, `TimerSettime()`, `TimerTimeout()`
TimerDestroy(),
TimerDestroy_r()

Destroy a process timer

Synopsis:

```c
#include <sys/neutrino.h>

int TimerDestroy( timer_t id );
int TimerDestroy_r( timer_t id );
```

Arguments:

- `id` The ID of the timer that you want to destroy, as returned by `TimerCreate()`.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These kernel calls remove a previously created timer specified by `id`. The timer is removed from the active system timer list and returned to the list of available timers.

The `TimerDestroy()` and `TimerDestroy_r()` functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `timer_delete()`.

If a timeout is pending when `TimerDestroy()` removes the timer, the timer is removed without being activated.

Blocking states

These calls don’t block.
Returns:

The only difference between these functions is the way they indicate errors:

TimerDestroy()

If an error occurs, -1 is returned and errno is set. Any other value returned indicates success.

TimerDestroy_r()

EOK is returned on success. This function does NOT set errno. If an error occurs, any value in the Errors section may be returned.

Errors:

EINVAL The timer specified by id doesn’t exist.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

timer_delete(), TimerCreate()
TimerInfo(), TimerInfo_r()

Get information about a timer

Synopsis:

```c
#include <sys/neutrino.h>

int TimerInfo( pid_t pid,
    timer_t id,
    int flags,
    struct _timer_info* info );

int TimerInfo_r( pid_t pid,
    timer_t id,
    int flags,
    struct _timer_info* info );
```

Arguments:

- **pid**: The process ID that you’re requesting the timer information for.
- **id**: The ID of the timer, as returned by `TimerCreate()`.
- **flags**: Supported flags are:
  - `_NTO_TIMER_SEARCH` — if this flag is specified and the timer ID doesn’t exist, return information on the next timer ID. This provides a mechanism to discover all of the timers in the process.
  - `_NTO_RESET_OVERRUNS` — reset the overrun count to zero in the `_timer_info` structure.
- **info**: A pointer to a `_timer_info` structure where the function can store the information about the specified timer. For more details, see “struct _timer_info,” below.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

These kernel calls get information about a previously created timer specified by \textit{id}, and stores the information in the buffer pointed to by \textit{info}.

The \texttt{TimerInfo()} and \texttt{TimerInfo\_r()} functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling \texttt{timer\_getexpstatus()}, \texttt{timer\_getoverrun()}, or \texttt{timer\_gettime()}.

\textbf{struct \_timer\_info}

The \texttt{\_timer\_info} structure pointed to by \textit{info} contains at least these members:

- \texttt{uint32\_t flags}
  
  One or more of these bit flags:

  \begin{verbatim}
  \_NTO\_TI\_ACTIVE
  The timer is active.
  \_NTO\_TI\_ABSOLUTE
  The timer is waiting for an absolute time to occur; otherwise, the timer is relative.
  \_NTO\_TI\_EXPIRED
  The timer has expired.
  \end{verbatim}

- \texttt{int32\_t tid}
  The thread to which the timer is directed (0 if it’s directed to the process).

- \texttt{int32\_t notify}
  The notify type.

- \texttt{clockid\_t clockid}
  The type of clock used.
The number of overruns.

The event dispatched when the timer expires.

Time when the timer was started.

Time remaining before the timer expires.

For more information, see the description of TimerCreate().

Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

TimerInfo() The ID of the timer that the information is for. If an error occurs, -1 is returned and errno is set.

TimerInfo_r() The ID of the timer that the information is for. This function does NOT set errno. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EINV AL The timer specified by id doesn’t exist.

ESRCH The process specified by pid doesn’t exist.
TimerInfo(), TimerInfo_r() © 2007, QNX Software Systems GmbH & Co. KG.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
sigevent, timer_getexpstatus(), timer_getoverrun(), timer_gettime(), TimerCreate()
## Synopsis:

```c
#include <sys/neutrino.h>

int TimerSettime( timer_t id,  
                 int flags,  
                 const struct _itimer * itime,  
                 struct _itimer * oitime );

int TimerSettime_r( timer_t id,  
                    int flags,  
                    const struct _itimer * itime,  
                    struct _itimer * oitime );
```

## Arguments:

- **id**
  The ID of the timer whose an expiration date you want to set, as returned by `TimerCreate()`.

- **flags**
  The only supported flag is `TIMER_ABSTIME`. If specified, then `nsec` represents an “absolute” expiration date in nanoseconds from the Unix Epoch, 00:00:00 January 1, 1970 UTC. If the date specified has already passed, then the expiration event is delivered immediately.
  If the flag isn’t specified, `nsec` represents a “relative” expiration period that’s offset from the given clock’s current system time in nanoseconds.

- **itime**
  A pointer to a `_itimer` structure that specifies the expiration date. For detailed information, see “Expiration date,” below.

- **oitime**
  NULL, or a pointer to a `_itimer` structure where the function can store the interval timer period (i.e. previous amount of time left before the timer was to have expired), or zero if the timer was disarmed at the time of the call.
  The previous interval timer period is also stored in the `interval_nsec` member.
**TimerSettime()**, **TimerSettime_r()** © 2007, QNX Software Systems GmbH & Co. KG.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `TimerSettime()` and `TimerSettime_r()` kernel calls set the expiration time of the timer specified by `id`.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `timer_gettime()` or `timer_settime()`.

**Expiration date**

The expiration is specified by the `itime` argument. The `_itimer` structure contains at least the following members:

```
uint64_t nsec
```

The expiration time to set.

```
uint64_t interval_nsec
```

The interval reload time.

If the `nsec` member of `itime` is zero, then the timer is disarmed.

If the `interval_nsec` member of `itime` is nonzero, then it specifies a repeat rate which is added to the timer once the `nsec` period has expired. Subsequently, the timer is automatically rearmed, causing it to become repetitive with a period of `interval_nsec`.

If the timer is already armed when you call `TimerSettime()`, this call discards the previous setting and sets a new setting.

If the event notification specified by `TimerCreate()` has a `sigev_code` of `SI_TIMER`, then at most one event is queued. In this case, if an
event is pending from a previous timer when the timer fires again, a
timer overrun occurs. You can use the TimerInfo() kernel call to
obtain the number of overruns that have occurred on this timer.

Blocking states

This call doesn’t block.

Returns:

The only difference between these functions is the way they indicate
errors:

TimerSettime()
If an error occurs, -1 is returned and errno is set. Any other
value returned indicates success.

TimerSettime_r()
EOK is returned on success. This function does NOT set errno.
If an error occurs, any value in the Errors section may be
returned.

Errors:

EINVAL The timer specified by id doesn’t exist.

EFAULT A fault occurred when the kernel tried to access itime or
overtime.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
**TimerSettime(), TimerSettime_r()** © 2007, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

*timer_gettime(), timer_settime(), TimerCreate(), TimerInfo()*
Synopsis:

```c
#include <sys/neutrino.h>

int TimerTimeout(clockid_t id,  
                  int flags,  
                  const struct sigevent * notify,  
                  const uint64_t * ntime,  
                  uint64_t * otime);

int TimerTimeout_r(clockid_t id,  
                    int flags,  
                    const struct sigevent * notify,  
                    const uint64_t * ntime,  
                    uint64_t * otime);
```

Arguments:

- **id** The type of timer to implement the timeout; one of:
  - CLOCK_REALTIME — This is the standard POSIX-defined clock. Timers based on this clock should will wake up the processor if it’s in a power-saving mode.
  - CLOCK_SOFTTIME — This clock is only active when the processor is not in a power-saving mode. For example, an application using a CLOCK_SOFTTIME timer to sleep wouldn’t wake up the processor when the application was due to wake up. This will allow the processor to enter a power-saving mode. While the processor isn’t in a power-saving mode, CLOCK_SOFTTIME behaves the same as CLOCK_REALTIME.
  - CLOCK_MONOTONIC — This clock always increases at a constant rate and can’t be adjusted.

- **flags** Flags that specify which states the timeout applies to. For the list and description of applicable states, see the section “Timeout states.”
**TimerTimeout(), TimerTimeout_r()**

GmbH & Co. KG.

- notify: A pointer to a `sigevent` structure that contains the event to act on when the timeout expires. See “Event types” for the list of recommended event types.

- ntime: The timeout (in nanoseconds).

- otime: NULL, or a pointer to a location where the function can store the time remaining in the sleep.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `TimerTimeout()` and `TimerTimeout_r()` kernel calls set a timeout on any kernel blocking state.

These functions are identical except in the way they indicate errors. See the Returns section for details.

Instead of using these kernel calls directly, consider calling `timer_timeout()`.

These blocking states are entered as a result of the following kernel calls:

<table>
<thead>
<tr>
<th>Call</th>
<th>Blocking state</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InterruptWait()</code></td>
<td><code>STATE_INTR</code></td>
</tr>
<tr>
<td><code>MsgReceivev()</code></td>
<td><code>STATE_RECEIVE</code></td>
</tr>
<tr>
<td><code>MsgSendv()</code></td>
<td><code>STATE_SEND</code> or <code>STATE_REPLY</code></td>
</tr>
</tbody>
</table>

*continued...*
TimerTimeout(),
TimerTimeout_r()

<table>
<thead>
<tr>
<th>Call</th>
<th>Blocking state</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignalSuspend()</td>
<td>STATE_SIGSUSPEND</td>
</tr>
<tr>
<td>SignalWaitinfo()</td>
<td>STATE_SIGWAITINFO</td>
</tr>
<tr>
<td>SyncCondvarWait()</td>
<td>STATE_CONDVAR</td>
</tr>
<tr>
<td>SyncMutexLock()</td>
<td>STATE_MUTEX</td>
</tr>
<tr>
<td>SyncSemWait()</td>
<td>STATE_SEM</td>
</tr>
<tr>
<td>ThreadJoin()</td>
<td>STATE_JOIN</td>
</tr>
</tbody>
</table>

Timeout states

You can specify which states the timeout should apply to via a bitmask passed in the flags argument. The bits are defined by the following constants:

_NTO_TIMEOUT_CONDVAR
  Timeout on STATE_CONDVAR.

_NTO_TIMEOUT_JOIN
  Timeout on STATE_JOIN.

_NTO_TIMEOUT_INTR
  Timeout on STATE_INTR.

_NTO_TIMEOUT_MUTEX
  Timeout on STATE_MUTEX.

_NTO_TIMEOUT_RECEIVE
  Timeout on STATE_RECEIVE.

_NTO_TIMEOUT_REPLY
  Timeout on STATE_REPLY.

_NTO_TIMEOUT_SEM
  Timeout on STATE_SEM.
TimerTimeout(), TimerTimeout_r()

_CLOCK_REALTIME,

_Timeout on _STATE_SEND.

_Timeout on _STATE_SIGSUSPEND.

_Timeout on _STATE_SIGWAITINFO.

For example, to set a timeout on _MsgSendv(), specify:

_Timeout_SEND | _TIMEOUT_REPLY

Once a timeout is specified using _TimerTimeout(), it’s armed and
released under the following conditions:

Armed The kernel attempts to enter a blocking state specified
in _flags.

Released One of the above kernel calls completed without
blocking, or the kernel call blocks but unblocks before
the timeout expires, or the timeout expires.

_TimerTimeout() always operates on a one-shot basis. When one of the
above kernel calls returns (or is interrupted by a signal), the timeout
request is removed from the system. Only one timeout per thread may
be in effect at a time. A second call to _TimerTimeout(), without calling
one of the above kernel functions, replaces the existing timeout on
that thread. A call with _flags set to zero ensures that a timeout won’t
occur on any state. This is the default when a thread is created.

Always call _TimerTimeout() just before the function that you wish to
timeout. For example:

... event.sigev_notify = SIGEV_UNBLOCK;

_timeout = 10×1000000000;

_TimerTimeout( CLOCK_REALTIME,
TimerTimeout(),
TimerTimeout_r()

... _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY,
        &event, &timeout, NULL );
MsgSendv( coid, NULL, 0, NULL, 0 );
...

If the signal handler is called between the calls to TimerTimeout() and
MsgSendv(), the TimerTimeout() values are saved during the signal
handler and then are restored when the signal handler exits.

EventTypes

If the timeout expires, the kernel acts upon the event specified in the
sigevent structure pointed to by the notify argument. We
recommend the following event types in this case:

- SIGEV_SIGNAL
- SIGEV_SIGNAL_CODE
- SIGEV_SIGNAL_THREAD
- SIGEV_PULSE
- SIGEV_UNBLOCK
- SIGEV_INTR

Only SIGEV_UNBLOCK guarantees that the kernel call unblocks. A
signal may be ignored, blocked, or accepted by another thread, and a
pulse can only unblock a MsgReceivev(). If you pass NULL for event,
SIGEV_UNBLOCK is assumed. In this case, a timed out kernel call
returns failure with an error of ETIMEDOUT.

MsgSendv() doesn’t unblock on SIGEV_UNBLOCK if the server has
already received the message via MsgReceivev() and has specified
_NTO_CHF_UNBLOCK in the flags argument to its ChannelCreate()
call. In this case, it’s up to the server to do a MsgReplyv().
The timeout

The type of timer used to implement the timeout is specified with the \textit{id} argument.

The timeout:

- Is specified by the \textit{ntime} argument (the number of nanoseconds).
- Is relative to the current time (when \textit{TimerTimeout()} is called), unless \textit{flags} includes \texttt{TIMER_ABSTIME}, which makes the timeout occur at the absolute time set in \textit{ntime}.
- Occurs on a clock tick (see \textit{ClockPeriod()}) so the actual wakeup time is a minimum of:
  
  \[
  \left( \frac{\text{ntime}}{\text{size of timer tick}} \right) \text{ nanoseconds}
  \]

If you specify a resolution that amounts to 1.7 timer ticks, you’ll wake up in at least 1.7 timer ticks.

If you don’t wish to block for any time, you can pass a NULL for \textit{ntime}, in which case no timer is used, the event is assumed to be \texttt{SIGEV_UNBLOCK}, and an attempt to enter a blocking state as set by \textit{flags} immediately returns with \texttt{ETIMEDOUT}. Although a questionable practice, you can use it to poll potential blocking kernel calls. For example, you can poll for messages using \textit{MsgReceivev()} with an immediate timeout. A much better approach is to use multiple threads and have one block waiting for messages.

If you set \textit{flags} to \texttt{_NTO_TIMEOUT_NANOSLEEP}, then these calls block in the \texttt{STATE_NANOSLEEP} state until the timeout (or a signal that unblocks the thread) occurs. You can use this to implement an efficient kernel sleep as follows:

\begin{verbatim}
TimerTimeout( CLOCK_REALTIME, _NTO_TIMEOUT_NANOSLEEP,
              NULL, ntime, otime);
\end{verbatim}

If \textit{otime} isn’t NULL and the sleep is unblocked by a signal, it contains the time remaining in the sleep.

The actual timeout that occurred is returned in \textit{otime}. The resolution of \textit{otime} for both \textit{timer_timeout()} and \textit{TimerTimeout()} functions is in
nanoseconds. The difference, however, for otime in these two functions is in the format. For timer_timeout(), the otime is a pointer to timespec structure with two integers, whereas for TimerTimeout(), the pointer is of uint64_t type.

Blocking states

These calls don’t block unless you specify _NTO_TIMEOUT_NANOSLEEP in flags. In this case, the calls block as follows:

STATE_NANOSLEEP

The calling thread blocks for the requested time period.

Returns:

The only difference between these functions is the way they indicate errors:

TimerTimeout()

The previous flags. If an error occurs, -1 is returned and errno is set.

TimerTimeout_r()

The previous flags. This function does NOT set errno. If an error occurs, the negative of a value from the Errors section is returned.

Errors:

EAGAIN All kernel timer entries are in use.
EFAULT A fault occurred when the kernel tried to access ntime, otime, or notify.
EINVAL The call was interrupted by a signal.
EINVAL Invalid timer value id.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

sigevent, timer_timeout(), TimerCreate(), TimerInfo()
times()

Get time-accounting information

Synopsis:

```c
#include <sys/times.h>

clock_t times( struct tms* buffer );
```

Arguments:

`buffer` A pointer to a `tms` structure where the function can store
the time-accounting information. For information about the
`tms` structure, see below.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is
usually included automatically.

Description:

The `times()` function stores time-accounting information in the
structure pointed to by `buffer`. The type `clock_t` and the `tms`
structure are defined in the `<sys/times.h>` header file.

The `tms` structure contains at least the following members:

```c

clock_t tms_utime
    The CPU time charged for the execution of user instructions of
    the calling process.

clock_t tms_stime
    The CPU time charged for execution by the system on behalf of
    the calling process.

clock_t tms_cutime
    The sum of the `tms_utime` and `tms_cutime` values of the child
    processes.
```

September 10, 2007
**clock_t tms_cstime**

The sum of the `tms_stime` and `tms_cstime` values of the child processes.

All times are in CLK_TCK’ths of a second. CLK_TCK is defined in the `<time.h>` header file. A CLK_TCK is the equivalent of:

```c
#define sysconf( _SC_CLK_TCK )
```

The times of a terminated child process are included in the `tms_cutime` and `tms_cstime` elements of the parent when a `wait()` or `waitpid()` function returns the process ID of this terminated child. If a child process hasn’t waited for its terminated children, their times aren’t included in its times.

---

**Returns:**

The elapsed real time, in clock ticks, of kernel uptime.

---

The value returned may overflow the possible range of type `clock_t`.

---

**Examples:**

```c
/*
 * The following program executes the program
 * specified by argv[1]. After the child program
 * is finished, the cpu statistics of the child are
 * printed.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/times.h>

int main( int argc, char **argv )
{
    struct tms childtim;

    system( argv[1] );
    times( &childtim );
    printf( "system time = %d\n", childtim.tms_cstime );
    printf( "user time = %d\n", childtim.tms_cutime );
    return EXIT_SUCCESS;
}
```
Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`clock_gettime()`
timespec

Time-specification structure

Synopsis:

```
#include <time.h>

struct timespec {
    time_t tv_sec;
    long tv_nsec;
}
```

Description:

The `timespec` structure specifies a time in seconds and nanoseconds. The members include:

- `tv_sec` The number of seconds. If specifying an absolute time, this member is the number of seconds since 1970.
- `tv_nsec` The number of nanoseconds.

Classification:

POSIX 1003.1

See also:

`nsec2timespec()`, `timespec2nsec()`
timespec2nsec()  
Convert a timespec structure to nanoseconds

Synopsis:
#include <time.h>

_uint64 timespec2nsec( const struct timespec* ts );

Arguments:
    ts    A pointer to the timespec that you want to convert to nanoseconds.

Library:
    libc
    Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:
The timespec2nsec() function converts the number of seconds and nanoseconds in the timespec structure pointed to by ts into nanoseconds.

Returns:
The number of nanoseconds.

Classification:
QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

September 10, 2007 C Library — T to Z 3517
See also:

`nsec2timespec()`, `timespec`
timezone

The number of seconds by which the local time zone is earlier than UTC

Synopsis:

```c
#include <time.h>

long int timezone;
```

Description:

This global variable holds the number of seconds by which the local time zone is earlier than Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time). Whenever you call a time function, `tzset()` is called to set the variable, based on the current time zone.

Classification:

POSIX 1003.1 XSI

See also:

`daylight, tzname, tzset()`

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
Synopsis:

```c
#include <time.h>
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
    long int tm_gmtoff;
    const char * tm_zone;
};
```

Description:

The `tm` structure describes the calendar time. The members of this structure include:

- `tm_sec`: Seconds after the minute, in the range [0,61], allowing for leap seconds.
- `tm_min`: Minutes after the hour, in the range [0,59].
- `tm_hour`: Hours after midnight, in the range [0,23].
- `tm_mday`: Day of the month, in the range [1,31].
- `tm_mon`: Months since January, in the range [0,11].
- `tm_year`: Years since 1900.
- `tm_wday`: Days since Sunday, in the range [0,6].
- `tm_yday`: Days since January 1, in the range [0,365], allowing for leap years.
- `tm_isdst`: Daylight saving time flag.
- `tm_gmtoff`: Offset from UTC — see `setlocale()`.
String for the time zone name.

Classification:

ANSI, POSIX 1003.1

See also:

asctime(), gmtime(), gmtime_r(), localtime(), localtime_r(), mktime(), setlocale(), strftime(), wcsftime()
tmpfile(), tmpfile64()  
Create a temporary file

Synopsis:

```c
#include <stdio.h>

FILE* tmpfile( void );
FILE* tmpfile64( void );
```

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `tmpfile()` and `tmpfile64()` functions create a temporary file and opens a corresponding FILE stream. The file is automatically removed when it’s closed or when the program terminates. The file is opened in update mode (as in `fopen()`’s w+ mode).

If the process is killed between file creation and unlinking, a permanent file may be left behind.

---

When a stream is opened in update mode, both reading and writing may be performed. However, writing may not be followed by reading without an intervening call to the `fflush()` function, or to a file-positioning function (fseek(), fsetpos(), rewind()). Similarly, reading may not be followed by writing without an intervening call to a file-positioning function, unless the read resulted in end-of-file.

Returns:

A pointer to the stream of the temporary file, or NULL if an error occurs (errno is set).
**tmpfile(), tmpfile64()**

**Errors:**

- **EACCESS** The calling process doesn’t have permission to create the temporary file.
- **EMFILE** The calling process already has already used OPEN_MAX file descriptors.
- **ENFILE** The system already has the maximum number of files open.
- **EROF$$** The filesystem for the temporary file is read-only.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

static FILE *TempFile;

int main( void )
{
    TempFile = tmpfile();
    ...
    fclose( TempFile );

    /* The temporary file will be removed when we exit. */
    return EXIT_SUCCESS;
}
```

**Classification:**

`tmpfile()` is ANSI, POSIX 1003.1; `tmpfile64()` is Large-file support

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

fopen(), fopen64(), freopen(), freopen64(), tempnam(), tmpnam()
tmpnam()

Generate a unique string for use as a filename

Synopsis:

```c
#include <stdio.h>

char* tmpnam( char* buffer );
```

Arguments:

- `buffer` NULL, or a pointer to a buffer where the function can store the filename. If `buffer` isn’t NULL, the buffer must be at least `L_tmpnam` bytes long.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tmpnam()` function generates a unique string that’s a valid filename and that’s not the same as the name of an existing file.

The `tmpnam()` function generates up to `TMP_MAX` unique file names before it starts to recycle them.

The generated filename is prefixed with the first accessible directory contained in:

- The `TMPDIR` environment variable
- The temporary file directory `P_tmpdir` (defined in `<stdio.h>`)  
- The `_PATH_TMP` constant (defined in `<paths.h>`)  

If all of these paths are inaccessible, `tmpnam()` attempts to use `/tmp` and then the current working directory.

The generated filename is stored in an internal buffer; if `buffer` is NULL, the function returns a pointer to this buffer; otherwise, `tmpnam()` copies the filename into `buffer`.  

September 10, 2007 C Library — T to Z 3525
Subsequent calls to `tmpnam()` reuse the internal buffer. If `buffer` is NULL, you might want to duplicate the resulting string. For example,

```c
char *name1, *name2;
name1 = strdup( tmpnam( NULL ) );
name2 = strdup( tmpnam( NULL ) );
```

**Returns:**

A pointer to the generated filename for success, or NULL if an error occurs (`errno` is set).

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    char filename[L_tmpnam];
    FILE *fp;

    tmpnam( filename );
    fp = fopen( filename, "w+b" );

    ... ...
    fclose( fp );
    remove( filename );

    return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
</tbody>
</table>

...continued...
Caveats:

The `tmpnam()` function *isn’t thread-safe* if you pass it a NULL buffer.

This function only creates pathnames; the application must create and remove the files.

It’s possible for another thread or process to create a file with the same name between when the pathname is created and the file is opened.

See also:

`tempnam()`, `tmpfile()`
tolower()  
Convert a character to lowercase

Synopsis:
```
#include <ctype.h>

int tolower( int c );
```

Arguments:
- `c` The character that you want to convert.

Library:
- `libc`
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `tolower()` function converts `c` to a lowercase letter, if `c` represents an uppercase letter.

Returns:
The corresponding lowercase letter when the argument is an uppercase letter; otherwise, the original character is returned.

Examples:
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char chars[] = {
    'A',
    '5',
    '$',
    'Z'
};

#define SIZE sizeof( chars ) / sizeof( char )

int main( void )
{
    int i;
```
for( i = 0; i < SIZE; i++ ) {
    printf( "%c", tolower( chars[ i ] ) );
} 
printf( "\n" );
return EXIT_SUCCESS;
}

produces the output:

a 5 $ z

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), strlwr(), strupr(), toupper()
toupper() — Convert a character to uppercase

Synopsis:

```
#include <ctype.h>

int toupper( int c );
```

Arguments:

- **c**   The character that you want to convert.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `toupper()` function converts `c` to an uppercase letter, if `c` represents a lowercase letter.

Returns:

- The corresponding uppercase letter when the argument is a lowercase letter; otherwise, the original character is returned.

Examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char chars[] = {
    'a',
    '5',
    '$',
    'z'
};

#define SIZE sizeof( chars ) / sizeof( char )

int main( void )
{
    int i;
```
for( i = 0; i < SIZE; i++ ) {
    printf( "%c", toupper( chars[ i ] ) );
}
printf( "\n" );
return EXIT_SUCCESS;
}

produces the output:

A 5 $ Z

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), strlwr(),strupr(), tolower()
towctrans()  
Convert a wide character in a specified manner

Synopsis:

```c
#include <wctype.h>

wint_t towctrans( wint_t wc,
          wctrans_t category );
```

Arguments:

- **wc**: The wide character that you want to convert.
- **category**: How you want to convert the character; get this by calling `wctrans()`.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `towctrans()` function converts `wc`, using the mapping described by `category`. The following functions are equivalent:

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent <code>wctrans()</code> call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tolower( wc )</code></td>
<td><code>towctrans( wc, wctrans(&quot;tolower&quot;) )</code></td>
</tr>
<tr>
<td><code>toupper( wc )</code></td>
<td><code>towctrans( wc, wctrans(&quot;toupper&quot;) )</code></td>
</tr>
</tbody>
</table>

Returns:

The corresponding converted wide character when the argument is valid; otherwise, the original wide character.
towctrans()

Errors:

EINV AL The conversion descriptor in category is invalid.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

wctrans()

“Character manipulation functions” and “Wide-character functions” in Library Reference Summary
towlower()  © 2007, QNX Software Systems GmbH & Co. KG.
Convert a wide character to lowercase

Synopsis:

```
#include <wctype.h>

wint_t towlower( wint_t wc );
```

Arguments:

`wc`  The wide character that you want to convert.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `towlower()` function converts `wc` to a lowercase letter, if `wc` represents an uppercase letter.

Returns:

The corresponding lowercase letter when the argument is an uppercase letter; otherwise, the original wide character is returned.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

“Character manipulation functions” and “Wide-character functions” in Library Reference Summary
towupper()  
Convert a wide character to uppercase

Synopsis:

```c
#include <wctype.h>

wint_t towupper( wint_t wc );
```

Arguments:

- `wc`: The wide character that you want to convert.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `towupper()` function converts `wc` to an uppercase letter if `wc` represents a lowercase letter.

Returns:

The corresponding uppercase letter when the argument is a lowercase letter; otherwise, the original wide character is returned.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

3536  C Library — T to Z  September 10, 2007
See also:

“Character manipulation functions” and “Wide-character functions” in *Library Reference Summary*
Synopsis:

```c
#include <traceparser.h>

struct tracebuf {  
    struct traceheader h;  
    struct traceevent data [TRACELEMENTS];  
};
```

Library:

`libtraceparser`

Description:

The `tracebuf` structure describes one buffer in the circular linked list of tracebuffers. The structure includes the following members.

- `h`: Header info structure; see `traceheader`.
- `data`: `TRACELEMENTS` variable or const?
- `TRACELEMENTS` variable or const?

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Yes/No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:
Synopsis:

```c
#include <sys/neutrino.h>
#include <sys/trace.h>

int TraceEvent( int mode, ...
```

Library:

libc

Description:

The `TraceEvent()` function controls all stages of the instrumentation process including initialization, starting, stopping, filter control and event insertion. These stages are broadly grouped into the following categories:

- Buffer and execution control
- Fast/wide mask configuration
- Static rules filter configuration
- Dynamic rules filter configuration
- User-generated trace events

This description also includes these sections:

- Argument descriptions
- `class` argument descriptions

Filter and mask settings are made regardless of the previous settings. Use care to not accidentally override or delete previous configurations.
Buffer and execution control

These modes control the buffer set up as well as start and stop logging.

_NTO TRACE_ALLOCBUFFER, _NTO TRACE_DEALLOCBUFFER

\[
\text{TraceEvent}\left(\_\text{NTO TRACE_ALLOCBUFFER}, \text{uint } \text{bufnum}, \text{ void** } \text{linkliststart}\right) \\
\text{TraceEvent}\left(\_\text{NTO TRACE_DEALLOCBUFFER}\right)
\]

The allocation option creates and initializes the internal circular link list of trace buffers. The created and initialized trace buffers hold the emitting trace events.

\[\text{bufnum}\]
Number of buffers to allocate.

\[\*\text{linkliststart}\]
Physical address of the beginning of the circular link list of allocated trace buffers.

Allocated trace buffers can store 1024 simple trace events.

\[\text{If your application calls this mode, it must run as root.}\]

The deallocation option deallocates all of the previously allocated trace buffers. All events stored inside of the trace buffers are lost.

_NTO TRACE_FLUSHBUFFER

\[
\text{TraceEvent}\left(\_\text{NTO TRACE_FLUSHBUFFER}\right)
\]

Forces flashing of the buffer regardless of the trace event number it contains.

_NTO TRACE_QUERYEVENTS

\[
\text{TraceEvent}\left(\_\text{NTO TRACE_QUERYEVENTS}\right)
\]

Returns the number of simple trace events that’s currently stored in the trace buffer.
TraceEvent()

_NTO_TRACE_START, _NTO_TRACE_STARTNOSTATE, _NTO_TRACE_STOP

    TraceEvent ( _NTO_TRACE_START )
    TraceEvent ( _NTO_TRACE_STARTNOSTATE )
    TraceEvent ( _NTO_TRACE_STOP )

Starts/stops the instrumentation process. The event stream containing the trace events is opened/closed.

The _NTO_TRACE_START and _NTO_TRACE_STARTNOSTATE options are the same except the latter suppresses the initial system state information (names of processes and thread IDs.)

_NTO_TRACE_SETRINGMODE

    TraceEvent ( _NTO_TRACE_SETRINGMODE )

Sets a ring mode of the internal circular link list. When an external application uses this mode, the kernel stores all events in a circular fashion inside the link list without flushing them. The maximum capturing time (without history overwriting) is determined by the number of allocated buffers, as well as by the number of generated trace events.

_NTO_TRACE_SETLINEARMODE

    TraceEvent ( _NTO_TRACE_SETLINEARMODE )

Sets a default linear mode of the internal circular link list. When using this mode, every filled-up buffer is captured and flushed immediately.

Fast/wide mask configuration

These modes control the operation of the fast/wide mask. For more information about this mask, see the Filtering chapter in this guide.

Currently, only the kernel call related classes are affected by the fast/wide modes. In fast mode, only two of the most important kernel call arguments and two of the most important kernel call return values are intercepted. See the Kernel call arguments and return values appendix for the list of the intercepted arguments and return values.
**TraceEvent()**

_NTO_TRACE_SETALCLASSESFAST, _NTO_TRACE_SETALCLASSESWIDE_

```
TraceEvent(_NTO_TRACE_SETALCLASSESFAST)
TraceEvent(_NTO_TRACE_SETALCLASSESWIDE)
```

Sets the fast/wide emitting mode for all classes and events.

_NTO_TRACE_SETCLASSFAST, _NTO_TRACE_SETCLASSWIDE_

```
TraceEvent(_NTO_TRACE_SETCLASSFAST, int class)
TraceEvent(_NTO_TRACE_SETCLASSWIDE, int class)
```

Sets the fast/wide emitting mode for all events within the specified `class`.

_NTO_TRACE_SETEVENTFAST, _NTO_TRACE_SETEVENTWIDE_

```
TraceEvent(_NTO_TRACE_SETEVENTFAST, int class, int event)
TraceEvent(_NTO_TRACE_SETEVENTWIDE, int class, int event)
```

Sets the fast/wide emitting mode for the specified `event` for the specified `class`.

**Static rules filter configuration**

These modes control the operation of the static rules filter. For more information about this filter, see the Filtering chapter in this guide.

_NTO_TRACE_ADDALCLASSES, _NTO_TRACE_DELALCLASSES_

```
TraceEvent(_NTO_TRACE_ADDALCLASSES)
TraceEvent(_NTO_TRACE_DELALCLASSES)
```

Emit/suppress trace events for all classes and events.

_NTO_TRACE_ADDCLASS, _NTO_TRACE_DELCLASS_

```
TraceEvent(_NTO_TRACE_ADDCLASS, class)
TraceEvent(_NTO_TRACE_DELCLASS, class)
```

Emit/suppress all trace events from a specific `class`.

_NTO_TRACE_ADDEVENT, _NTO_TRACE_DELEVENT_

```
TraceEvent(_NTO_TRACE_ADDEVENT, class, event)
TraceEvent(_NTO_TRACE_DELEVENT, class, event)
```

Emit/suppress a trace `event` from a specific `class`.
**TraceEvent()**

© 2007, QNX Software Systems GmbH & Co. KG.

_NTO_TRACE_SETCLASSPID, _NTO_TRACE_CLRCCLASSPID, _NTO_TRACE_SETCLASSTID, _NTO_TRACE_CLRCCLASSPID, _NTO_TRACE_SETCLASSPID, _NTO_TRACE_CLRCCLASSPID

TraceEvent(_NTO_TRACE_SETCLASSPID, int class, pid_t pid)
TraceEvent(_NTO_TRACE_CLRCCLASSPID, int class)
TraceEvent(_NTO_TRACE_SETCLASSTID, int class, pid_t pid, tid_t tid)
TraceEvent(_NTO_TRACE_CLRCCLASSPID, int class)

Emits/suppresses all events from a specified process ID (and thread ID).

_NTO_TRACE_SETEVENTPID, _NTO_TRACE_CLREVENTPID, _NTO_TRACE_SETEVENTTID, _NTO_TRACE_CLREVENTTID,

TraceEvent(_NTO_TRACE_SETEVENTPID, int class, int event, pid_t pid)
TraceEvent(_NTO_TRACE_CLREVENTPID, int class, int event)
TraceEvent(_NTO_TRACE_SETEVENTTID, int class, int event, pid_t pid, tid_t tid)
TraceEvent(_NTO_TRACE_CLREVENTTID, int class, int event)

Emits/suppresses a specific event for a specified process ID (and thread ID.).

**Dynamic rules filter configuration**

These modes control the operation of the dynamic rules filter. For more information about this filter, see the Filtering chapter in this guide.

**EventHandler Data Access**

The access to the trace event information from within the event handler can be done using members of the `event_data_t`.

The valid layout of the `event_data_t` structure (declared in `sys/trace.h`) is as follow:

```c
/* event data filled by an event handler */
typedef struct
{
    __traceentry header; /* same as traceevent header */
    _Uint32t* data_array; /* initialized by the user */
    _Uint32t el_num; /* number of elements returned */
    void* area; /* user data */
    _Uint32t feature_mask; /* bits indicate valid features */
    _Uint32t feature[_NTO_TRACE_FI_NUM]; /* feature array */
} event_data_t;
```
The bits of the member feature_mask are related to any additional feature (argument) that could be accessed inside the event handler. All standard data arguments, the ones that correspond to the data arguments of the trace-event, are delivered without changes within array data_array[]. If any particular bit of the feature_mask is set to value equal to 1, then, the feature corresponding to this bit can be accessed within array feature[]. Otherwise, the feature should not be accessed. For example, if the expression:

```
feature_mask & _NTO_TRACE_FMPID
```

has its logical value equal to TRUE, then, the additional feature corresponding to identifier _NTO_TRACE_FMPID (PID) can be accessed as:

```
my_pid = feature[_NTO_TRACE_FIPID];
```

For every additional feature there have been provided two constants:

- _NTO_TRACE_FM*** - feature parameter masks
- _NTO_TRACE_FI*** - feature index parameters

to check and to access the given feature.

**_NTO_TRACE_ADDEVENTHANDLER, _NTO_TRACE_DELEVENTHANDLER**

```
TraceEvent(_NTO_TRACE_ADDEVENTHANDLER,
class, event,
int (*event_hdlr) (event_data_t*),
event_data_t* data_struct)
```

**_NTO_TRACE_ADDCLASSEVHANDLER, _NTO_TRACE_DELCLASSEVHANDLER**

```
TraceEvent(_NTO_TRACE_ADDCLASSEVHANDLER,
class,
int (*event_hdlr) (event_data_t*),
event_data_t* data_struct)
```

```
TraceEvent(_NTO_TRACE_DELCLASSEVHANDLER,
class)
```
Attaches/deletes the event handler for a specified class, where:

\[ event\_hdlr \]
Pointer to the event handler.

\[ data\_struct \]
Pointer to the data structure \texttt{event\_data\_t}.

In order to emit an event data, a dynamic filter (event handler) has to return 1. If both types of the dynamic filters (event handler and class event handler) are applicable to a particular event, the event is emitted if both event handlers return 1.

**User-generated trace events**

These modes control the insertion of “fake” events into the event stream.

\texttt{NTO\_TRACE\_INSERTEVENT}

\begin{verbatim}
TraceEvent( \_NTO\_TRACE\_INSERTEVENT, int head, int stamp, int data0, int data1)
\end{verbatim}

Inserts a generic, “real” event into the event stream. It’s powerful but because the API doesn’t do any of the interpretation for you, this function should be used with care by advanced users only. The data-interpretation program must be modified to properly interpret the event.

The arguments are:

\[ head \]
Header of the trace event.

\[ stamp \]
Time stamp.

\[ data0 \]
Data d0.

\[ data1 \]
Data d1.

\texttt{NTO\_TRACE\_INSERTSUSEREVENT, NTO\_TRACE\_INSERTCUSEREVENT, NTO\_TRACE\_INSERTUSRSTREVENT}

\begin{verbatim}
TraceEvent( \_NTO\_TRACE\_INSERTSUSEREVENT, int event, int data0, int data1)
TraceEvent( \_NTO\_TRACE\_INSERTCUSEREVENT, int event, unsigned * buf, unsigned len)
TraceEvent( \_NTO\_TRACE\_INSERTUSRSTREVENT, int event, const char * str)
\end{verbatim}
The `len` argument represents the number of integers in `buf`.

These modes insert user-created events into the event stream. Because the API handles details such as timestamping, they’re reasonably easy to use.

- `_NTO_TRACE_INSERTSUSEREVENT`
  Simple user event.
- `_NTO_TRACE_INSERTCUSEREVENT`
  Complex user event.
- `_NTO_TRACE_INSERTUSRSTREVENT`
  User string event.

The arguments are:

- `event` User defined event code. The value should be between `_NTO_TRACE_USERFIRST` and `_NTO_TRACE_USERLAST`.
- `str` Null terminated string.
- `data0` Data d0.
- `data1` Data d1.

The `TraceEvent()` function controls all stages of the instrumentation process such as initialization, starting, execution control, and stopping. These stages consist of the following activities:

- creating an internal circular linked list of trace buffers
- initializing filters
- turning on or off the event stream
- deallocating the internal circular linked list of trace buffers
TraceEvent() function accepts any number of arguments grouped logically as follows:

\[
\text{TraceEvent}(\text{mode} [, \text{class} [, \text{event}]] [, p_1 [, p_2 [, p_3 \ldots [, p_n]]]])
\]

Here’s a description of the arguments:

**mode**

Specifies a control action (compulsory).

You’ll find a description for each `mode` argument listed below in the `mode` argument descriptions section.

Some `mode` arguments require additional arguments; see the table of argument hierarchy for details. Valid arguments are:

- `_NTO_TRACE_ADDALLCLASSES`
- `_NTO_TRACE_ADDCLASS`
- `_NTO_TRACE_ADDEVENT`
- `_NTO_TRACE_ADDEVENTHANDLER`
- `_NTO_TRACE_ALLOCBUFFER`
- `_NTO_TRACE_CLRCLASSPID`
- `_NTO_TRACE_CLRCLASSSTID`
- `_NTO_TRACE_CLREVENTPID`
- `_NTO_TRACE_CLREVENTTID`
- `_NTO_TRACE_DEALLOCBUFFER`
- `_NTO_TRACE_DELALLCLASSES`
- `_NTO_TRACE_DELCLASS`
- `_NTO_TRACE_DELEVENT`
- `_NTO_TRACE_DELEVENTHANDLER`
- `_NTO_TRACE_FLUSHBUFFER`
- `_NTO_TRACE_INSERTCUSEREVENT`
- `_NTO_TRACE_INSERTEVENT`
- `_NTO_TRACESUSEREVENT`
- `_NTO_TRACE_INSERTUSRSTREVENT`
- `_NTO_TRACE_QUERYEVENTS`
class

You’ll find a description for each class argument listed below in the class argument descriptions section. Some class arguments may require additional arguments; see the table of argument hierarchy for details.

Valid arguments are:

_NTO_TRACE_CONTROL
_NTO_TRACE_INT
_NTO_TRACE_INTENTER
_NTO_TRACE_INTEXIT
_NTO_TRACE_KERCALL
_NTO_TRACE_KERCALLENDER
_NTO_TRACE_KERCALLEXIT
_NTO_TRACE_PROCESS
_NTO_TRACE_THREAD
_NTO_TRACE_VTHREAD
event

Redirects the control action specified by the mode and class towards a trace event within the class.

You’ll find a description for each event argument listed below in the event argument descriptions section. Some event arguments may require additional arguments; see the table of argument hierarchy for details.

The following table shows the valid event arguments for a particular class:

<table>
<thead>
<tr>
<th>If the value of class is:</th>
<th>Then a valid event argument is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_TRACE_CONTROL</td>
<td>_NTO_TRACE_CONTROLTIME</td>
</tr>
<tr>
<td>_NTO_TRACE_INT,</td>
<td>a logical interrupt vector number</td>
</tr>
<tr>
<td>_NTO_TRACE_INTENTER,</td>
<td></td>
</tr>
<tr>
<td>_NTO_TRACE_INTEXIT</td>
<td></td>
</tr>
<tr>
<td>_NTO_TRACE_KERCALL,</td>
<td>A valid _<em>KER</em>* keyword from &lt;sys/kercalls.h&gt; (such as __KER_MSG_SENDV.)</td>
</tr>
<tr>
<td>_NTO_TRACE_KERCALLENTER,</td>
<td></td>
</tr>
<tr>
<td>_NTO_TRACE_KERCALLEXIT</td>
<td></td>
</tr>
<tr>
<td>_NTO_TRACE_PROCESS</td>
<td>_NTO_TRACE_PROCCREATE,</td>
</tr>
<tr>
<td></td>
<td>_NTO_TRACE_PROCCREATE_NAME,</td>
</tr>
<tr>
<td></td>
<td>_NTO_TRACE_PROCDESTROY,</td>
</tr>
<tr>
<td></td>
<td>_NTO_TRACE_PROCDESTROY_NAME</td>
</tr>
</tbody>
</table>

continued...
If the value of `class` is: Then a valid `event` argument is:

| _NTO_TRACE_THREAD       | _NTO_TRACE_THCONDEVAR,  |
| _NTO_TRACE_THCREATE,    |
| _NTO_TRACE_THDEAD,      |
| _NTO_TRACE_THDESTROY,   |
| _NTO_TRACE_THINTR,      |
| _NTO_TRACE_THJOIN,      |
| _NTO_TRACE_THMUTEX,     |
| _NTO_TRACE_THNANOSLEEP, |
| _NTO_TRACE_THNET_REPLY, |
| _NTO_TRACE_THNET_SEND,  |
| _NTO_TRACE_THREADY,     |
| _NTO_TRACE_THRECEIVE,   |
| _NTO_TRACE_THREPLY,     |
| _NTO_TRACE_THRUNNING,   |
| _NTO_TRACE_THSEM,       |
| _NTO_TRACE_THSEND,      |
| _NTO_TRACE_THSIGSUSPEND,|
| _NTO_TRACE_THSIGWAITINFO,|
| _NTO_TRACE_THSTACK,     |
| _NTO_TRACE_THSTOPPED,   |
| _NTO_TRACE_THWAITCTX,   |
| _NTO_TRACE_THWAITPAGE,  |
| _NTO_TRACE_THWAITTHREAD | continued...            |
### TraceEvent()

**If the value of class is:**  
**Then a valid event argument is:**  

<table>
<thead>
<tr>
<th>Class</th>
<th>Event Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NTO_TRACE_VTHREAD</td>
<td>_NTO_TRACE_VTHCONDVAR, _NTO_TRACE_VTHCREATE, _NTO_TRACE_VTHDEAD, _NTO_TRACE_VTDestroy, _NTO_TRACE_VTHJOIN, _NTO_TRACE_VTHJOIN, _NTO_TRACE_VTHMUTEX, _NTO_TRACE_VTHNANOSLEEP, _NTO_TRACE_VTHNANOSLEEP, _NTO_TRACE_VTHNET_REPLY, _NTO_TRACE_VTHNET_SEMAPHORE, _NTO_TRACE_VTHREADY, _NTO_TRACE_VTHREADY, _NTO_TRACE_VTHRECEIVE, _NTO_TRACE_VTHRECEIVE, _NTO_TRACE_VTHREPLY, _NTO_TRACE_VTHREPLY, _NTO_TRACE_VTHRUNNING, _NTO_TRACE_VTHRUNNING, _NTO_TRACE_VTHEM, _NTO_TRACE_VTHEM, _NTO_TRACE_VTHSEND, _NTO_TRACE_VTHSEND, _NTO_TRACE_VTHSIGSUSPEND, _NTO_TRACE_VTHSIGSUSPEND, _NTO_TRACE_VTHSIGWAITINFO, _NTO_TRACE_VTHSIGWAITINFO, _NTO_TRACE_VTHSTACK, _NTO_TRACE_VTHSTACK, _NTO_TRACE_VTHSTOPPED, _NTO_TRACE_VTHSTOPPED, _NTO_TRACE_VTHWAITCTX, _NTO_TRACE_VTHWAITCTX, _NTO_TRACE_VTHWAITPAGE, _NTO_TRACE_VTHWAITPAGE, _NTO_TRACE_VTHWAITTHREAD</td>
</tr>
</tbody>
</table>

### Argument descriptions

Specifies any additional parameters that are required to perform the desired control action.

The following are the generic arguments used for the \textit{TraceEvent() modes}. Mode-specific arguments accompany the mode description:

- **mode**  
  The control action. The mode is always the first argument in the \textit{TraceEvent()} function. Depending upon the value of mode, further arguments may be necessary. The...
description of what each mode does appears earlier in this section. Examples of the mode include:

-_NTO_TRACE_ALLOCBUFFER,
-_NTO_TRACE_SETCLASSFAST.

**class**
The category of events. All the events are logically grouped into several classes. A list of valid classes is given in class argument descriptions, later in this section.

**event**
The event. Because the events are grouped by class, the event must be a member of the class in order to be valid. A list of events can be found in the Kernel Call Arguments and Return Values chapter in this guide.

**pid**
Process ID to be registered.

**tid**
Thread ID to be registered.

### class argument descriptions

The class argument may be one of the following:

- **-_NTO_TRACE_CONTROL**
  Specifies the set of control events (i.e. time-overflow event) that’re used by the communication protocol between the microkernel and tracelogger.

- **-_NTO_TRACE_INTENTER, _NTO_TRACE_INEXIT**
  Specifies the set of interrupt entry/exit events.

- **-_NTO_TRACE_KERCALLENTER, _NTO_TRACE_KERCALLEXIT**
  Specifies the set of kernel call entry/exit events.

- **-_NTO_TRACE_PROCESS**
  Specifies the set of events associated with process creation and destruction.
TraceEvent()

_NTO_TRACE_THREAD,
_NTO_TRACE_VTHREAD

Specifies the set of class arguments that contain thread (or virtual thread) state changes, and create or destroy events.

There are also “pseudo classes” offered as a convenience:

_NTO_TRACE_KERCALL

Specifies all of the kernel call events:
_NTO_TRACE_KERCALLENTER and
_NTO_TRACE_KERCALLEXIT.

_NTO_TRACE_INT

Specifies all of the interrupt events: _NTO_TRACE_INTENTER and _NTO_TRACE_INTEXIT.

Returns:

If mode is set to _NTO_TRACE_QUERYEVENTS
Number of events in the buffer, or -1 if an error occurs (errno is set).

If mode isn’t set to _NTO_TRACE_QUERYEVENTS
0 for success, or -1 if an error occurs (errno is set).

Errors:

ECANCELED The requested action has been canceled.
EFAULT Bad internal trace buffer address. The requested action has been specified out of order.
ENOMEM Insufficient memory to allocate the trace buffers.
ENOSUP The requested action isn’t supported.
EPERM The application doesn’t have permission to perform the action.
Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

Caveats:

You can call `TraceEvent()` from an interrupt/event handler. However, not all trace modes are valid in this case. The valid trace modes are:

- `_NTO_TRACE_INSERTSUSEREVENT`
- `_NTO_TRACE_INSERTCUSEREVENT`
- `_NTO_TRACE_INSERTUSRSTREVENT`
- `_NTO_TRACE_INSERTEVENT`
- `_NTO_TRACE_STOP`
- `_NTO_TRACE_STARTNOSTATE`
- `_NTO_TRACE_START`

See also:

`InterruptAttach()`, `InterruptHookTrace()`
**Synopsis:**

```
#include <traceparser.h>

struct traceheader {
    struct tracebuf * next;
    uint32_t flags;
    uint32_t reserved1;
    uint32_t num_events;
    struct timeval start;
    struct timeval end;
    traceevent_t * begin_ptr;
    traceevent_t * end_ptr;
    traceevent_t volatile * head_ptr;
    traceevent_t volatile * tail_ptr;
    uint32_t max_events;
    struct intrspin spin;
    paddr_t baseaddr;
    uint32_t reserved2;
};
```

**Library:**

libtraceparser

**Description:**

The `traceheader` structure describes header information. The structure includes the following members:

- **next**    The size of the entire tracebuffer @@@size or actual?.
- **flags**   Opaque variable; version information, event mask.
- **reserved1** Opaque variable; not yet used.
- **num_events** The number of events in the buffer.
- **start**   Start of the trace.
end            End of the trace.
begin_ptr      @@ @.
end_ptr        @@ @.
head_ptr       @@ @.
tail_ptr       @@ @.

max_events     The maximum number of events that occur before
               the buffer flushes.

spin           Spin lock.

base_addr      The base address of the tracebuffer.

reserved2      Opaque variable; not yet used.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:
traceparser()

Execute a parsing procedure with user data

Synopsis:

```
#include <sys/traceparser.h>

extern int traceparser (  
    struct traceparser_state * stateptr,  
    void * userdata,  
    const char * filename );
```

Library:

libtraceparser

Description:

The `traceparser()` function starts the parsing procedure `filename`. It also executes the user defined callback functions and passes the `userdata` to it. The `stateptr` argument is an opaque structure obtained from `traceparser_init()`.

Returns:

0 Success

-1 Failure; `errno` is set. See also `traceparser_get_info()` for further details.

Classification:

QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

3558 C Library — T to Z September 10, 2007
See also:

`traceparser_get_info()`, `traceparser_init()`
traceparser_cs()

Attach a callback

Synopsis:

```c
#include <traceparser.h>

int traceparser_cs (  
   struct traceparser_state * stateptr,  
   void * userdata,  
   tracep_callb_func_t funcptr,  
   unsigned class,  
   unsigned event );
```

Library:

libtraceparser

Description:

The `traceparser_cs()` function attaches one callback function, specified by the pointer `funcptr`, to one particular `event`, from one particular `class`. The user data (`userdata`) is passed to the attached callback function upon execution. The `stateptr` is an opaque structure obtained from `traceparser_init()`.

Returns:

- 0 Success; a pointer to the event
- -1 Failure; `errno` is set. See also `traceparser_get_info()` for further details.

Classification:

QNX Neutrino

Safety

- Cancellation point: No
- Interrupt handler: No

continued…
traceparser_cs()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

traceparser_get_info(), traceparser_init()
traceparser_cs_range() © 2007, QNX Software Systems GmbH & Co. KG.

Attach a callback to a range of events

Synopsis:

```c
#include <traceparser.h>

int traceparser_cs_range (  
    struct traceparser_state * stateptr,  
    void * userdata,  
    tracep_callb_func_t funcptr,  
    unsigned class,  
    unsigned firstevent,  
    unsigned lastevent );
```

Library:

libtraceparser

Description:

The `traceparser_cs_range()` function attaches one callback function, given by the pointer `funcptr`, to a range of events from `firstevent` through to `lastevent` inclusive, from one particular `class`. The user data (`userdata`) is passed to the registered callback function (`funcptr`) upon execution. The `stateptr` is an opaque structure obtained from `traceparser_init()`.

Returns:

0 Success; a pointer to the list of events.

-1 Failure; `errno` is set. See also `traceparser_get_info()` for further details

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

continued...
traceparser_cs_range()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

traceparser_get_info(), traceparser_init()
traceparser_debug()  
Set the traceparser debug modes

Synopsis:

```c
#include <traceparser.h>

int traceparser_debug (  
    struct traceparser_state * stateptr,  
    FILE * streamptr,  
    unsigned flags );
```

Library:

libtraceparser

Description:

The `traceparser_debug()` function sets the debug modes of the traceparser module. The `streamptr` argument is a pointer to the debug output stream; `flags` specifies the debug category. The `stateptr` is an opaque structure obtained from `traceparser_init()`.

Debug flags

The following is a list of the arguments that may be used for `flags` and the debug level for each:

- `_TRACEPARSER_DEBUG_ALL`
  - Everything.
- `_TRACEPARSER_DEBUG_ERRORS`
  - Critical errors only.
- `_TRACEPARSER_DEBUG_EVENTS`
  - Row input events only.
- `_TRACEPARSER_DEBUG_HEADER`
  - Header information only.
- `_TRACEPARSER_DEBUG_NONE`
  - No debugging.
traceparser_debug()

_TRACEPARSER_DEBUG_SYSPAGE
    Syspage data only.

Returns:

> 0   Success; a pointer to the event

-1   Failure; errno is set. See also traceparser_get_info() for further details.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

traceparser_get_info(), traceparser_init()
traceparser_destroy()  © 2007, QNX Software Systems GmbH & Co. KG.
Destroys a traceparser state structure

Synopsis:

```c
#include <traceparser.h>

void traceparser_destroy ( 
        struct traceparser_state ** stateptr );
```

Library:

`libtraceparser`

Description:

The `traceparser_destroy()` function destroys a previously initialized
traceparser state structure, `stateptr`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`traceparser_get_info()`, `traceparser_init()`
traceparser get info()

Get information about traceparser

Synopsis:

```c
#include <sys/traceparser.h>

void * traceparser_get_info ( 
    struct traceparser_state * stateptr, 
    info_modes_t infomodes, 
    unsigned * len);
```

Library:

libtraceparser

Description:

The traceparser_get_info() function gets information related to the state of the traceparser. The infomodes argument may be any of the constants shown below and are defined in traceparser.h.

The len argument is a pointer to the size of the return buffer. When specified, its contents are changed to indicate the size of the return. This is primarily for the TRACEPARSER_SYSPAGE and TRACEPARSER_HEADER_KEYWORDS modes but it’ll work for all the modes. For most of the modes, len may be NULL.

The stateptr is an opaque structure obtained from traceparser_init().

User info modes for info_modes

The following are valid user info modes; see the list below for others.

<table>
<thead>
<tr>
<th>Value for info_modes and Pointer to return data type, cast as</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_TRACEPARSER_INFO_SYSPAGE syspage_entry</td>
<td>Returns a pointer to the syspage entry.</td>
</tr>
</tbody>
</table>

continued...
**traceparser_get_info()**

© 2007, QNX Software Systems GmbH & Co. KG.

### Value for info_modes and Pointer to return data type, cast as void

<table>
<thead>
<tr>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
</table>
| Returns a dereferenced pointer; 1 if the endian conversion has been applied, 0 if no conversion has been performed. | _TRACEPARSER_INFO_ENDIAN_CONV 

unsigned |
| Returns the class or event numerical value of the currently executed callback function. The numerical values are considered opaque and should be used only for other traceparser functions. | _TRACEPARSER_INFO_NOW_CALLBACK_CLASS 

unsigned |
| _TRACEPARSER_INFO_NOW_CALLBACK_EVENT 

unsigned |
| Returns the class or event numerical value of the previously executed callback function. | _TRACEPARSER_INFO_PREV_CALLBACK_CLASS 

unsigned |
| _TRACEPARSER_INFO_PREV_CALLBACK_EVENT 

unsigned |
| Returns the value of the previously executed callback function. | _TRACEPARSER_INFO_DEBUG 

unsigned |
| Returns the traceparser error level. It must be used to determine traceparser library related errors. (See *sys/traceparser.h* for a list of the returned error values.) | _TRACEPARSER_INFO_ERROR 

unsigned |

### Other valid user info modes

The following modes return a pointer to the header field of the buffer. All data types are *void*.

- _TRACEPARSER_INFO_FILE_NAME
- _TRACEPARSER_INFO_DATE
- _TRACEPARSER_INFO_VER_MAJOR
- _TRACEPARSER_INFO_VER_MINOR
traceparser_get_info()

- TRACEPARSER_INFO_LITTLE_ENDIAN
- TRACEPARSER_INFO_BIG_ENDIAN
- TRACEPARSER_INFO_MIDDLE_ENDIAN
- TRACEPARSER_INFO_ENCODING
- TRACEPARSER_INFO_BOOT_DATE
- TRACEPARSER_INFO_CYCLES_PER_SEC
- TRACEPARSER_INFO_CPU_NUM
- TRACEPARSER_INFO_SYSNAME
- TRACEPARSER_INFO_NODENAME
- TRACEPARSER_INFO_SYS_RELEASE
- TRACEPARSER_INFO_SYS_VERSION
- TRACEPARSER_INFO_MACHINE
- TRACEPARSER_INFO_SYSPAGE_LEN

Returns:
A pointer to void
Success.
Null Failure; errno is set. See also the _TRACEPARSER_ERROR section of this function for further details.

Classification:
QNX Neutrino

Safety
Cancellation point No
continued…
traceparser_get_info()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

traceparser_get_info(), traceparser_init()
traceparser_init()

Initialize or get the state of the traceparser library

Synopsis:

```c
#include <sys/traceparser.h>

struct traceparser_state * traceparser_init ( 
    struct traceparser_state * stateptr );
```

Library:

libtraceparser

Description:

The `traceparser_init()` function initializes the state of the traceparser library. To initialize the library, execute the function with the `stateptr` argument as null; the function returns the initialized state structure.

The `traceparser_state` structure is an opaque structure for use by the other SAT functions.

Returns:

A pointer to a valid initialized state structure

- Success.
- `NULL` Failure; `errno` is set. See also `traceparser_get_info()` for further details.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

September 10, 2007
See also:

traceparser_get_info(), traceparser_init()
truncatem

Truncate a file to a specified length

Synopsis:

```c
#include <unistd.h>

int truncate( const char* path,  
              off_t length );
```

Arguments:

- `path`   The path name of the file that you want to truncate.
- `length` The new size of the file.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `truncate()` function causes the regular file named by `path` to have a size of `length` bytes.

The effect of `truncate()` on other types of files is unspecified. If the file previously was larger than `length`, the extra data is lost. If it was previously shorter than `length`, bytes between the old and new lengths are read as zeroes. The process must have write permission for the file.

If the request would cause the file size to exceed the soft file size limit for the process, the request fails and the implementation generates the SIGXFSZ signal for the process.

This function doesn’t modify the file offset for any open file descriptions associated with the file. On successful completion, if the file size is changed, `truncate()` marks for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.
truncate()

Returns:

0  Success.

-1  An error occurred; errno is set.

Errors:

EACCES  A component of the path prefix denies search permission, or write permission is denied on the file.

EFAULT  The path argument points outside the process’s allocated address space.

EFBIG  The length argument was greater than the maximum file size.

EINTR  A signal was caught during execution.

EINVAL  The length argument is invalid, or the path argument isn’t an ordinary file.

EIO  An I/O error occurred while reading from or writing to a filesystem.

EISDIR  The named file is a directory.

ELOOP  Too many symbolic links were encountered in resolving path.

EMFILE  The maximum number of file descriptors available to the process has been reached.

EMULTIHOP Components of path require hopping to multiple remote machines and filesystem type doesn’t allow it.

ENAMETOOLONG  The length of the specified pathname exceeds PATH_MAX bytes, or the length of a component of the pathname exceeds NAME_MAX bytes.
ENFILE  Additional space couldn’t be allocated for the system file table.
ENOENT  A component of path doesn’t name an existing file or path is an empty string.
ENOLINK The path argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR A component of the path prefix of path isn’t a directory.
EROFS  The named file resides on a read-only filesystem.

Classification:
POSIX 1003.1 XSI

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chmod(), fcntl(), ftruncate(), open()
**ttyname()**

Get a fully qualified pathname for a file

**Synopsis:**

```c
#include <unistd.h>

char *ttyname( int filedes );
```

**Arguments:**

- `fd`es: A file descriptor that's associated with the file whose name you want to get.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ttyname()` function returns a pointer to a static buffer that contains a fully qualified pathname associated with the file associated with `fd`es.

**Returns:**

A pointer to the pathname for `fd`es, or NULL if an error occurred (`errno` is set).

**Errors:**

- `EBADF` The `fd`es argument is invalid.
- `ENOSYS` The `ttyname()` function isn’t implemented for the filesystem specified by `fd`es.
- `ENOTTY` Not a tty.
Examples:

/*
 * The following program prints out the name
 * of the terminal associated with stdin.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( void )
{
    if( isatty( 0 ) ) {
        printf( "%s\n", ttyname( 0 ) );
    } else {
        printf( "\n" );
        return EXIT_SUCCESS;
    }
}

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

ctermid(), setsid(), ttyname_r()
ttynamel r() — Get a fully qualified pathname for a file

Synopsis:

```c
#include <unistd.h>

int ttynamel r( int fildes,
               char* name,
               size_t namesize );
```

Arguments:

- `fildes` A file descriptor that’s associated with the file whose name you want to get.
- `name` A pointer to a buffer where the function can store the path name.
- `namesize` The size of the buffer.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ttynamel r()` function stores the null-terminated pathname of the terminal associated with the file descriptor `fildes` in the character array referenced by `name`. The array is `namesize` characters long and should have space for the name and the terminating NULL character.

Returns:

Zero for success, or an error number.

Errors:

- EBADF The `fildes` argument isn’t a valid file descriptor.
- ENOSYS The `ttynamel r()` function isn’t implemented for the filesystem specified by `fildes`.
ENOTTY  The *fildes* argument doesn’t refer to a tty.
ERANGE  The value of *namesize* is smaller than the length of the
        string to be returned, including the terminating null
        character.

Classification:

POSIX 1003.1 TSF

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

ctermid(), errno, setsid(), ttyname()
The abbreviations for the time zone for standard and daylight savings time

Synopsis:

```c
#include <time.h>

char *tzname[];
```

Description:

This global variable holds the standard abbreviations for the time zone and the time zone when daylight saving time is in effect. Whenever you call a time function, `tzset()` is called to set the values in the array, based on the current time zone.

Classification:

POSIX 1003.1 XSI

See also:

`daylight`, `timezone`, `tzset()`

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
Synopsis:

```c
#include <time.h>

void tzset( void );
```

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `tzset()` function sets the global variables `daylight`, `timezone` and `tzname` according to the value of the `TZ` environment variable, or to the value of the `_CS_TIMEZONE` configuration string if `TZ` isn’t set.

The global variables have the following values after `tzset()` is executed:

- `daylight`: Zero indicates that daylight saving time isn’t supported in the locale; a nonzero value indicates that daylight saving time is supported in the locale. This variable is cleared or set after a call to the `tzset()` function, depending on whether or not a daylight saving time abbreviation is specified in the `TZ` environment variable.

- `timezone`: The number of seconds that the local time zone is earlier than Coordinated Universal Time (UTC) (formerly known as Greenwich Mean Time (GMT)).

- `tzname`: A two-element array pointing to strings giving the abbreviations for the name of the time zone when standard and daylight saving time are in effect.

The time that you set on the computer with the `date` command reflects Coordinated Universal Time (UTC). The environment variable `TZ` is used to establish the local time zone. For more
information, see “Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void print_zone()
{
    char *tz;

    printf( "TZ: %s\n", (tz = getenv( "TZ" ))
     ? tz : "default EST5EDT" );
    printf( " daylight: %d\n", daylight );
    printf( " timezone: %ld\n", timezone );
    printf( " time zone names: %s %s\n", 
             tzname[0], tzname[1] );
}

int main( void )
{
    tzset();
    print_zone();
    setenv( "TZ", "PST8PDT", 1 );
    tzset();
    print_zone();
    return EXIT_SUCCESS;
}
```

produces the output:

```
TZ: default EST5EDT
    daylight: 1
    timezone: 18000
    time zone names: EST EDT
TZ: PST8PDT
    daylight: 1
    timezone: 28800
    time zone names: PST PDT
```

Classification:

POSIX 1003.1 XSI
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

- `ctime()`, `daylight`, `localtime()`, `localtime_r()`, `mktime()`, `strftime()`
- `timezone`, `tzname`

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino *User’s Guide*
**ualarm()**

Schedule an alarm

**Synopsis:**

```c
#include <unistd.h>

useconds_t ualarm( useconds_t usec,
                  useconds_t interval );
```

**Arguments:**

- `usec`: The number of microseconds that you want to elapse before the first alarm occurs, or 0 to cancel any previous request for an alarm.
- `interval`: The number of microseconds that you want to elapse before the subsequent alarms occur.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ualarm()` function causes the system to send the calling process a SIGALRM signal after `usec` microseconds of real-time have elapsed. The alarm is then sent every `interval` microseconds after that.

Processor scheduling delays may cause a delay between when the signal is sent and when the process actually handles it.

If `usec` is 0, any previous `ualarm()` request is canceled.

**Returns:**

- 0: There was no previous `ualarm()` request.
- -1: An error occurred (`errno` is set).

Any other value

The number of microseconds until the next scheduled SIGALRM.

© 2007, QNX Software Systems GmbH & Co. KG.
Errors:

EAGAIN All timers are in use; wait for a process to release one and try again.

Examples:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    useconds_t timeleft;

    printf( "Set the alarm and sleep\n" );
    ualarm( (useconds_t)( 10 * 1000 * 1000 ), 0 );
    sleep( 5 ); /* go to sleep for 5 seconds */

    /*
     * To get the time left before the SIGALRM is to arrive, one must cancel
     * the initial timer, which returns the amount of time it had remaining.
     */
    timeleft = ualarm( 0, 0 );
    printf( "Time left before cancel, and rearm: %ld\n",
            timeleft );

    /*
     * Start a new timer that kicks us when timeleft seconds have passed.
     */
    ualarm( timeleft, 0 );

    /*
     * Wait until we receive the SIGALRM signal; any signal kills us, though,
     * since we don’t have a signal handler.
     */
    printf( "Hanging around, waiting to exit\n" );
    pause();

    /* You’ll never get here. */
    return EXIT_SUCCESS;
}
```
**Classification:**

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

*alarm()* , *TimerAlarm()* , and *ualarm()* requests aren’t “stacked”; only a single SIGALRM generator can be scheduled with these functions. If the SIGALRM signal hasn’t been generated, the next call to *alarm()* , *TimerAlarm()* , or *ualarm()* reschedules it.

Don’t mix calls to *ualarm()* with *nanosleep()* , *sleep()* , *timer_create()* , *timer_delete()* , *timer_getoverrun()* , *timer_gettime()* , *timer_settime()* , or *usleep()*.

**See also:**

*alarm()* , *nanosleep()* , *sigaction()* , *sleep()* , *timer_create()* ,
*timer_delete()* , *timer_getoverrun()* , *timer_gettime()* , *timer_settime()* ,
*TimerAlarm()* , *usleep()*
**Description:**

UDP is a simple, unreliable datagram protocol that’s used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless and are normally used with the sendto() and recvfrom() calls, although you can also use the connect() call to fix the destination for future packets (in which case you can use the recv() or read() and send() or write() system calls).

UDP address formats are identical to those used by TCP. In particular, UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space; that is, a UDP port may not be “connected” to a TCP port. In addition, broadcast packets may be sent — assuming the underlying network supports this — by using a reserved broadcast address; this address is network-interface dependent.

You can use options at the IP transport level with UDP (see the IP protocol).

**Returns:**

A descriptor referencing the socket, or -1 if an error occurs (errno is set).

**Errors:**

EADDRINUSE You tried to create a socket with a port that has already been allocated.
EADDRNOTAVAIL
You tried to create a socket with a network address for which no network interface exists.

EISCONN
You tried to establish a connection on a socket that already has one, or to send a datagram with the destination address specified and the socket is already connected.

ENOBUS
The system ran out of memory for an internal data structure.

ENOTCON
You tried to send a datagram, but no destination address was specified and the socket hasn’t been connected.

See also:

IP protocol
connect(), getsockopt(), read(), recv(), recvfrom(), send(), sendto(), socket(), write()

RFC 768
**ultoa(), ulltoa()**

Convert an unsigned long integer into a string, using a given base

**Synopsis:**

```c
#include <stdlib.h>

char* ultoa( unsigned long int value,
              char* buffer,
              int radix);

char* ulltoa( unsigned long long value
              char* buffer,
              int radix);
```

**Arguments:**

- `value`  The value to convert into a string.
- `buffer` A buffer in which the function stores the string. The size of the buffer must be at least 33 bytes when converting values in base 2 (binary).
- `radix`  The base to use when converting the number. This value must be in the range:

  \[ 2 \leq \text{radix} \leq 36 \]

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ultoa()` and `ulltoa()` functions convert the unsigned binary integer `value` into the equivalent string in base `radix` notation, storing the result in the character array pointed to by `buffer`. A NUL character is appended to the result.
Returns:

A pointer to the result.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>

void print_value( unsigned long int value )
{
    int base;
    char buffer[33];

    for( base = 2; base <= 16; base = base + 2 )
        printf( "%2d %s\n", base, ultoa( value, buffer, base ) );
}

int main( void )
{
    print_value( (unsigned) 12765L );
    return EXIT_SUCCESS;
}
```

produces the output:

```
2 11000111011101
4 3013131
6 135033
8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

`ultoa()` is QNX 4; `ulltoa()` is Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

continued…
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`atoi()`, `atol()`, `itoa()`, `ltoa()`, `sscanf()`, `strtol()`, `strtoul()`, `utoa()`
Set the file-mode creation mask for the process

**Synopsis:**
```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask( mode_t cmask );
```

**Arguments:**
- `cmask` The new file-mode creation mask; that is, the permissions that you don’t want set when the process creates a file. The mask is a combination of these bits:

<table>
<thead>
<tr>
<th>Owner</th>
<th>Group</th>
<th>Others</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IRUSR</td>
<td>S_IREGRP</td>
<td>S_IROTH</td>
<td>Read</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>S_IREXGRP</td>
<td>S_IREXO</td>
<td>Read, write, execute/search. A bitwise inclusive OR of the other three constants. (S_IRWXU is OR of IRUSR, S_IRUSR and S_IXUSR.)</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>S_IREWGRP</td>
<td>S_IROWTH</td>
<td>Write</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>S_IXGRP</td>
<td>S_IXOTH</td>
<td>Execute/search</td>
</tr>
</tbody>
</table>

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `umask()` function sets the process’s file-mode creation mask to `cmask`, and returns the previous value of the mask. Only the file permission bits (as defined in `<sys/stat.h>`) are used.

The file-mode creation mask for the process is used when you call `creat()`, `mkdir()`, `mkfifo()`, and `open()`, to turn off permission bits in the
mode argument supplied. Bit positions set in cmask are cleared in the mode of the created file.

Returns:

The previous value of the file-mode creation mask.

Examples:

```c
/*
 * Set the umask to RW for owner, group; R for other
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void )
{
    mode_t omask;
    mode_t nmask;

    nmask = S_IRUSR | S_IWUSR | /* owner read write */
           S_IRGRP | S_IWGRP | /* group read write */
           S_IROTH; /* other read */
    omask = umask( nmask );
    printf( "Mask changed from %o to %o\n", omask, nmask );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`chmod()`, `creat()`, `mkdir()`, `mkfifo()`, `open()`, `stat()`
umount()

Unmount a filesystem

Synopsis:

```c
#include <sys/mount.h>

int umount( const char* dir, int flags);
```

Arguments:

- `dir` The filesystem that you want to unmount.
- `flags` Flags that control the operation. Currently, the only valid value for `flags` is:
  - `_MOUNT_FORCE` — force an unmount to occur.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `umount()` function sends a request to the server to unmount the path described by `dir`.

Returns:

-1 on failure.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

continued…
umount()

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mount()

Writing a Resource Manager in *Programmer’s Guide*
Synopsis:

```c
#include <gulliver.h>

void UNALIGNED_PUT16( uint16_t *loc, uint16_t num );
```

Arguments:

- `loc` The address where you want to write the value.
- `num` The value that you want to write.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `UNALIGNED_PUT16()` macro lets you write the value `num` at the misaligned address `loc` without faulting.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
UNALIGNED_PUT16()

Caveats:

UNALIGNED_PUT16() is implemented as a macro.

See also:

ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET32(), ENDIAN_RET64(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohl(),
ntohs(), UNALIGNED_PUT32(), UNALIGNED_PUT64(),
UNALIGNED_RET16(), UNALIGNED_RET32(),
UNALIGNED_RET64()
UNALIGNMENTS_PUT32()

Write a misaligned 32-bit value safely

Synopsis:

```c
#include <gulliver.h>

void UNALIGNMENTS_PUT32( uint32_t *loc,
                          uint32_t num );
```

Arguments:

- `loc` The address where you want to write the value.
- `num` The value that you want to write.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `UNALIGNMENTS_PUT32()` macro lets you write the value `num` at the misaligned address `loc` without faulting.

Classification:

- QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**UNALIGNED_PUT32()**

Caveats:

*UNALIGNED_PUT32()* is implemented as a macro.

See also:

*ENDIAN_BE16()*, *ENDIAN_BE32()*, *ENDIAN_BE64()*,
*ENDIAN_LE16()*, *ENDIAN_LE32()*, *ENDIAN_LE64()*,
*ENDIAN_RET32()*, *ENDIAN_RET64()*, *ENDIAN_SWAP16()*,
*ENDIAN_SWAP32()*, *ENDIAN_SWAP64()*, *htonl()*, *htons()*, *ntohl()*,
*ntohs()*, *UNALIGNED_PUT16()*, *UNALIGNED_PUT32()*,
*UNALIGNED_RET16()*, *UNALIGNED_RET32()*,
*UNALIGNED_RET64()*
UNALIGNED_PUT64()

Write a misaligned 64-bit value safely

Synopsis:

#include <gulliver.h>

void UNALIGNED_PUT64( uint64_t * loc, 
                       uint64_t num );

Arguments:

loc The address where you want to write the value.
num The value that you want to write.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The UNALIGNED_PUT64() macro lets you write the value num at the misaligned address loc without faulting.

This macro isn’t currently implemented.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
Caveats:

UNALIGNED_PUT64() is implemented as a macro.

See also:

ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET32(), ENDIAN_RET64(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohl(),
ntohs(), UNALIGNED_PUT16(), UNALIGNED_PUT32(),
UNALIGNED_RET16(), UNALIGNED_RET32(),
UNALIGNED_RET64()
UNALIGNED_RET16()
Access a misaligned 16-bit value safely

Synopsis:
```
#include <gulliver.h>

uint16_t UNALIGNED_RET16(const uint16_t *loc);
```

Arguments:
`loc` The address where you want to get the value from.

Library:
`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `UNALIGNED_RET16()` macro lets you access the misaligned 16-bit value pointed to by `loc` without faulting.

Returns:
The 16-bit value pointed to by `loc`.

Classification:
QNX Neutrino

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

`UNALIGNED_RET16()` is implemented as a macro.

See also:

`ENDIAN_BE16()`, `ENDIAN_BE32()`, `ENDIAN_BE64()`,
`ENDIAN_LE16()`, `ENDIAN_LE32()`, `ENDIAN_LE64()`,
`ENDIAN_RET32()`, `ENDIAN_RET64()`, `ENDIAN_SWAP16()`,
`ENDIAN_SWAP32()`, `ENDIAN_SWAP64()`, `htonl()`, `htons()`, `ntohl()`,
`ntohs()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`,
`UNALIGNED_PUT64()`, `UNALIGNED_RET32()`,
`UNALIGNED_RET64()`
UNALIGNED_RET32()
Access a misaligned 32-bit value safely

Synopsis:

```
#include <gulliver.h>

uint32_t UNALIGNED_RET32( const uint32_t *loc );
```

Arguments:

`loc` The address where you want to get the value from.

Library:

`libc`
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `UNALIGNED_RET32()` macro lets you access the misaligned 32-bit value pointed to by `loc` without faulting.

Returns:

The 32-bit value pointed to by `loc`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

UNALIGNED_RET32() is implemented as a macro.

See also:

ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET32(), ENDIAN_RET64(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohl(),
ntobs(), UNALIGNED_PUT16(), UNALIGNED_PUT32(),
UNALIGNED_PUT64(), UNALIGNED_RET16(),
UNALIGNED_RET32(), UNALIGNED_RET64()
UNALIGNED RET64()
Access a misaligned 64-bit value safely

Synopsis:

#include <gulliver.h>

uint64_t UNALIGNED_RET64( const uint64_t * loc );

Arguments:

loc The address where you want to get the value from.

Library:

libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The UNALIGNED_RET64() macro lets you access the misaligned 64-bit value pointed to by loc without faulting.

This macro isn’t currently implemented.

Returns:

The 64-bit value pointed to by loc.

Classification:

QNX Neutrino

Safety

Cancellation point No
Interrupt handler Yes
Signal handler Yes

continued...
UNALIGNED_RET64()

Safety

| Thread | Yes |

Caveats:

UNALIGNED_RET64() is implemented as a macro.

See also:

ENDIAN_BE16(), ENDIAN_BE32(), ENDIAN_BE64(),
ENDIAN_LE16(), ENDIAN_LE32(), ENDIAN_LE64(),
ENDIAN_RET32(), ENDIAN_RET64(), ENDIAN_SWAP16(),
ENDIAN_SWAP32(), ENDIAN_SWAP64(), htonl(), htons(), ntohs(),
ntohl(), UNALIGNED_PUT16(), UNALIGNED_PUT32(),
UNALIGNED_PUT64(), UNALIGNED RET16(),
UNALIGNED RET32()
uname()

Get information about the operating system

Synopsis:

```c
#include <sys/utsname.h>

int uname( struct utsname * name );
```

Arguments:

name A pointer to a `utsname` where the function can store the information; see below.

Library:

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `uname()` function stores information about the current operating system in the structure pointed to by the argument `name`.

The system name structure, `utsname`, is defined in `<sys/utsname.h>`, and contains at least the following structure members:

- `char* sysname` The name of the OS.
- `char* nodename` The name of this node.
- `char* release` The current release level.
- `char* version` The current version level.
- `char* machine` The hardware type.

Each of these items is a null-terminated character array.
uname()

Returns:

0  Success.
-1  An error occurred (errno is set).

Examples:

```
/*
 * The following program prints some information about the
 * system it's running on.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utsname.h>

int main( void )
{
    struct utsname sysinfo;

    if( uname( &sysinfo ) == -1 ) {
        perror( "uname" );
        return EXIT_FAILURE;
    }
    printf( "system name : %s\n", sysinfo.sysname );
    printf( "node name : %s\n", sysinfo.nodename );
    printf( "release name : %s\n", sysinfo.release );
    printf( "version name : %s\n", sysinfo.version );
    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

Safety

| Cancellation point | No |
| Interrupt handler  | No |
| Signal handler     | Yes |
| Thread             | Yes |
See also:

errno

uname in the Utilities Reference
ungetc()  © 2007, QNX Software Systems GmbH & Co. KG.

Push a character back onto an input stream

Synopsis:

```c
#include <stdio.h>

int ungetc( int c,
            FILE *fp );
```

Arguments:

- `c` is the character that you want to push back.
- `fp` is the stream you want to push the character back on.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `ungetc()` function pushes the character specified by `c` back onto the input stream pointed to by `fp`. This character will be returned the next time that you read from the stream. The pushed-back character is discarded if you call `fflush()` or a file-positioning function (`fseek()`, `fsetpos()`, or `rewind()`) before performing the next read operation.

Only one character (the most recent one) of pushback is guaranteed.

The `ungetc()` function clears the end-of-file indicator, unless the value of `c` is EOF.

Returns:

The character pushed back.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main( void )
```
ungetc()

```c
{
    FILE *fp;
    int c;
    long value;

    fp = fopen("file", "r");
    value = 0;
    c = fgetc(fp);
    while(isdigit(c)) {
        value = value*10 + c - '0';
        c = fgetc(fp);
    }
    ungetc(c, fp); /* put last character back */
    printf("Value=%.ld\n", value);
    fclose(fp);
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fopen(), getc(), getc_unlocked(), ungetwc()
**ungetwc()**

Push a wide character back onto an input stream

**Synopsis:**

```c
#include <wchar.h>

wint_t ungetwc( wint_t wc,
                FILE * fp );
```

**Arguments:**

- `c` The wide character that you want to push back.
- `fp` The stream you want to push the wide character back on.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `ungetwc()` function pushes the wide character specified by `wc` back onto the input stream pointed to by `fp`.

The pushed-back character will be returned the next time that you read from the stream but is discarded if you call `fflush()` or a file-positioning function (seek(), fsetpos(), or rewind()) before the next read operation is performed.

Only one character (the most recent one) of pushback is guaranteed.

The `ungetwc()` function clears the end-of-file indicator, unless the value of `wc` is WEOF.

**Returns:**

The character pushed back.
Errors:

EILSEQ  Invalid character sequence or wide character.

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fopen(), getwc(), ungetc()
UNIX-domain protocol family

Synopsis:

```c
#include <sys/socket.h>
#include <sys/un.h>

socket( AF_LOCAL,
       SOCK_STREAM,
       0 );

socket( AF_LOCAL,
       SOCK_DGRAM,
       0 );
```

Description:

The UNIX-domain protocol family provides local (on-machine or QNX-network) interprocess communication through the normal `socket()` mechanisms. The UNIX-domain family supports the `SOCK_STREAM` and `SOCK_DGRAM` socket types and uses filesystem pathnames for addressing.

Addressing

UNIX-domain addresses are variable-length filesystem pathnames of at most 104 characters. The `<sys/un.h>` include file defines this address:

```c
struct sockaddr_un {
    u_char sun_len;
    u_char sun_family;
    char sun_path[104];
};
```

Binding a name to a UNIX-domain socket with `bind()` causes a socket file to be created in the filesystem. This file isn’t removed when the socket is closed; you must use `unlink()` to remove the file.

You can use the macro `SUN_LEN()` (defined in `<sys/un.h>`) to calculate the length of UNIX-domain address, required by `bind()` and `connect()`. The `sun_path` field must be terminated by a NUL character to be used with `SUN_LEN()`, but the terminating NUL isn’t part of the address.
The UNIX-domain protocol family doesn’t support broadcast addressing or any form of “wildcard” matching on incoming messages. All addresses are absolute- or relative-pathnames of other UNIX-domain sockets. Normal filesystem access-control mechanisms are also applied when referencing pathnames (e.g. the destination of a `connect()` or `sendto()` must be writable).

**Protocols**

The UNIX-domain protocol family consists of simple transport protocols that support the SOCK_STREAM and SOCK_DGRAM abstractions. UNIX-domain sockets also support the communication of QNX file descriptors through the use of the `msg_control` field in the `msg` argument to `sendmsg()` and `recvmsg()`.

Any valid descriptor may be sent in a message. The file descriptor to be passed is described using a `struct cmsghdr` defined in the include file `<sys/socket.h>`. The type of the message is SCM_RIGHTS, and the data portion of the messages is an array of integers representing the file descriptors to be passed. The number of descriptors being passed is defined by the length field of the message; the length field is the sum of the size of the header plus the size of the array of file descriptors.

The received descriptor is a duplicate of the sender’s descriptor, as if it were created with a call to `dup()`. Descriptors awaiting delivery or purposely not received are automatically closed by the system when the destination socket is closed.

**LOCAL_CREDS**

There is one socket-level option for `setsockopt()` and `getsockopt()` available in the UNIX-domain. The LOCAL_CREDS option may be enabled on a SOCK_DGRAM or a SOCK_STREAM socket. This option provides a mechanism for the receiver to receive the credentials of the process as a `recvmsg()` message. The `msg_control` field in the `msghdr` structure points to a buffer that contains a `cmsghdr` structure followed by a variable length `sockcred` structure defined in `<sys/socket.h>` as follows:

```c
struct sockcred {
```
uid_t sc_uid; /* real user id */
uid_t sc_euid; /* effective user id */
gid_t sc_gid; /* real group id */
gid_t sc_egid; /* effective group id */
int sc_ngroups; /* number of supplemental groups */
gid_t sc_groups[1]; /* variable length */
}

The `SOCKCREDSIZE()` macro computes the size of the `sockcred` structure for a specified number of groups. The `cmsghdr` fields have the following values:

```c
    cmsg_len = sizeof(struct cmsghdr) + SOCKCREDSIZE(ngroups)
    cmsg_level = SOL_SOCKET
    cmsg_type = SCM_CREDS
```

See also:

`bind()`, `connect()`, `dup()`, `getsockopt()`, `recvmsg()`, `sendmsg()`, `sendto()`, `setsockopt()`, `socket()`, `unlink()`
**Synopsis:**

```c
#include <unistd.h>

int unlink( const char * path );
```

**Arguments:**

- `path`  
  The name of the file that you want to unlink.

**Library:**

- `libc`  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `unlink()` function removes a link to a file:

- If the `path` names a symbolic link, `unlink()` removes the link, but doesn’t affect the file or directory that the link goes to.

- If the `path` isn’t a symbolic link, `unlink()` removes the link and decrements the link count of the file that the link refers to.

If the link count of the file becomes zero, and no process has the file open, then the space that the file occupies is freed, and no one can access the file anymore.

If one or more processes have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed.

---

To remove a directory, call `rmdir()` or `remove()`.
Returns:

0        The operation was successful.
Nonzero  The operation failed (errno is set).

Errors:

EACCES   Search permission is denied for a component of path, or write permission is denied on the directory containing the link to be removed.

EBUSY    The directory named by path cannot be unlinked because it’s being used by the system or another process, and the target filesystem or resource manager considers this to be an error.

ENAMETOOLONG
          The path argument exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.

ENOENT   The named file doesn’t exist, or path is an empty string.

ENOSYS   The unlink() function isn’t implemented for the filesystem specified by path.

ENOTDIR  A component of path isn’t a directory.

EPERM    The file named by path is a directory, and either the calling process doesn’t have the appropriate privileges, or the target filesystem or resource manager prohibits using unlink() on directories.

EROFS    The directory entry to be unlinked resides on a read-only filesystem.
unlink()

Examples:

```c
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    if( unlink( "vm.tmp" ) ) {
        puts( "Error removing vm.tmp!" );
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

chdir(), chmod(), close(), errno, getcwd(), link(), mkdir(), open(), pathmgr_symlink(), pathmgr_unlink(), remove(), rename(), rmdir(), stat(), symlink()
unsetenv()

Remove an environment variable

Synopsis:

```c
#include <stdlib.h>

void unsetenv( const char* name );
```

Arguments:

`name` The name of the environment variable that you want to delete.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `unsetenv()` function removes the environment variable named `name` from the process’s environment.

Classification:

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
unsetenv()

Caveats:

The `unsetenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

`clearenv()`, `getenv()`, `putenv()`, `setenv()`
**Synopsis:**

```c
#include <unistd.h>

int usleep( useconds_t useconds );
```

**Arguments:**

`useconds` The number of microseconds that you want to process to sleep for. This must be less than 1,000,000.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `usleep()` function suspends the calling thread until `useconds` microseconds of realtime have elapsed, or until a signal that isn’t ignored is received. The time spent suspended could be longer than the requested amount due to the scheduling of other, higher-priority threads.

If `useconds` is 0, `usleep()` has no effect.

**Returns:**

0 Success.

-1 An error occurred (`errno` is set).

**Errors:**

EAGAIN No timer resources are available to satisfy the request.

EINV AL The `useconds` argument is too large.
Examples:

```c
/*
 * The following program sleeps for the
 * number of microseconds specified in argv[1].
 */
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char **argv )
{
    useconds_t microseconds;
    microseconds = (useconds_t)strtol( argv[1], NULL, 0 );
    if( usleep( microseconds ) == 0 ) {
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

alarm(), nanosleep(), sigaction(), sleep(), timer_create(),
timer_delete(), timer_getoverrun(), timer_gettime(), timer_settime(),
ualarm()}
**utime()**

Record the modification time for a file or directory

**Synopsis:**

```c
#include <sys/types.h>
#include <utime.h>

struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};

int utime( const char* path,
            const struct utimbuf* times );
```

**Arguments:**

- `path`  
  The path name for the file whose modification time you want to get or set.

- `times`  
  NULL, or a pointer to a `utimbuf` structure where the function can store the modification time.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `utime()` function records the modification time for the file or directory identified by `path`.

If the `times` argument is NULL, the access and modification times of the file or directory are set to the current time. The effective user ID of the process must match the owner of the file or directory, or the process must have write permission to the file or directory, or appropriate privileges in order to use the `utime()` function in this way.

If the `times` argument isn’t NULL, its interpreted as a pointer to a `utimbuf` structure, and the access and modification times of the file or directory are set to the values contained in the designated structure.
Only the owner of the file or directory, and processes with appropriate privileges are permitted to use the `utime()` function in this way. The access and modification times are taken from the `actime` and `modtime` fields in this structure.

**Returns:**

0  Success.
-1  An error occurred; `errno` is set.

**Errors:**

- **EACCES**  Search permission is denied for a component of `path`, or the `times` argument is NULL, and the effective user ID of the process doesn’t match the owner of the file, and write access is denied.
- **ENAMETOOLONG**  The argument `path` exceeds PATH_MAX in length, or a pathname component is longer than NAME_MAX.
- **ENOENT**  The specified `path` doesn’t exist, or `path` is an empty string.
- **ENOTDIR**  A component of `path` isn’t a directory.
- **EPERM**  The `times` argument isn’t NULL, and the calling process’s effective user ID has write access to the file but doesn’t match the owner of the file, and the calling process doesn’t have the appropriate privileges.
- **EROFS**  The named file resides on a read-only filesystem.

**Examples:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <utime.h>

int main( int argc, char *argv[] )
{
```

September 10, 2007
if( (utime( argv[1], NULL ) != 0) && (argc > 1) ) {
    printf( "Unable to set time for %s\n", argv[1] );
    return EXIT_SUCCESS;
}

Classification:

POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, futime()
utimes()

Set a file’s access and modification times

Synopsis:

```c
#include <sys/time.h>

int utimes( const char * __path,
            const struct timeval * __times );
```

Arguments:

__path The name of the files whose times you want to set.
__times NULL, or an array of timeval structures:

- The first array member represents the date and time of last access.
- The second member represents the date and time of last modification.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The utimes() function sets the access and modification times of the file pointed to by the __path argument to the value of the __times argument. This function allows time specifications accurate to the microsecond.

The times in the timeval structure are measured in seconds and microseconds since the Unix Epoch (00:00:00 January 1, 1970 Coordinated Universal Time (UTC)), although rounding toward the nearest second may occur.

If the __times argument is NULL, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file, or must have write access to the file or superuser privileges to use this call in this manner.
On completion, `utimes()` marks the time of the last file status change, `st_ctime`, for update.

**Returns:**

0     Success.

-1    An error occurred (`errno` is set).

**Errors:**

EACCES  Search permission is denied by a component of the path prefix; or the `__times` argument is NULL and the effective user ID of the process doesn’t match the owner of the file and write access is denied.

EFAULT   The `__path` or `__times` argument points to an illegal address.

EINVAL   The number of microseconds specified in one or both of the `timeval` structures pointed to by `__times` was greater than or equal to 1,000,000 or less than 0.

EIO       An I/O error occurred while reading from or writing to the filesystem.

ELOOP     Too many symbolic links were encountered in resolving `__path`.

EMULTIHOP Components of `__path` require hopping to multiple remote machines and the filesystem doesn’t allow it.

ENAMETOOLONG The length of the `__path` argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.


ENOLINK  The __path argument points to a remote machine and the link to that machine is no longer active.

ENOENT  A component of __path doesn’t name an existing file or __path is an empty string.

ENOTDIR A component of the path prefix isn’t a directory.

EPERM  The __times argument isn’t NULL and the calling process’s effective user ID has write access to the file but doesn’t match the owner of the file, and the calling process doesn’t have the appropriate privileges.

EROF S  The filesystem containing the file is read-only.

ENAMETOOLONG  
Path name resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

stat()
Synopsis:

```c
struct utmp {
    char ut_user[UT_NAMESIZE];
#define ut_name ut_user
    char ut_id[4];
    char ut_line[UT_LINESIZE];
    pid_t ut_pid;
    short ut_type;
    struct exit_status {
        short e_termination;
        short e_exit;
    } ut_exit;
    short ut_spare;
    time_t ut_time;
};
```

Description:

The `utmp` structure describes an entry in a user-information file. The members include:

- `ut_user` The user’s login name.
- `ut_id` The line number.
- `ut_line` The device name (console).
- `ut_pid` The process ID.
- `ut_type` The type of entry. The possible values are:
  - EMPTY
  - RUN_LVL
  - BOOT_TIME
  - OLD_TIME
  - NEW_TIME
  - INIT_PROCESS
  - LOGIN_PROCESS
- USER_PROCESS
- DEAD_PROCESS
- ACCOUNTING

\textit{ut\_exit}  The exit status of a process marked as DEAD\_PROCESS.

The structure \textit{exit\_status} includes at least the following members:

- \textit{e\_termination} — the termination status.
- \textit{e\_exit} — the exit status.

\textit{ut\_time}  The time that this entry was made.

\textbf{Classification:}

Unix

\textbf{See also:}

\textit{endutent()}, \textit{getutent()}, \textit{getutid()}, \textit{getutline()}, \textit{pututline()}, \textit{setutent()}, \textit{utmpname()}

\textit{login} in the \textit{Utilities Reference}
**utmpname()**
Change the name of the user-information file

**Synopsis:**
```c
#include <utmp.h>

void utmpname( char * __filename );
```

**Arguments:**

__filename  The new filename that you want to use.

**Library:**

libc
Use the -l c option to gcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).

**Description:**

The utmpname() function lets you change the name of the file examined from the default file (PATH_UTMP) to any other file. If the file doesn’t exist, this won’t be apparent until the first attempt to reference the file is made. This function doesn’t open the file. It just closes the old file if it’s currently open and saves the new file name.

**Files:**

__PATH_UTMP
Specifications the user information file.

**Classification:**

Unix
**utmpname()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*endutent(), getent(), getutid(), getutline(), pututline(), setutent(), utmp*

*login* in the *Utilities Reference*
Convert an unsigned integer into a string, using a given base

Synopsis:
```
#include <stdlib.h>

char* utoa( unsigned int value,  
            char* buffer,  
            int radix );
```

Arguments:
- **value**: The value to convert into a string.
- **buffer**: A buffer in which the function stores the string. The size of the buffer must be at least:
  \[ 8 \times \text{sizeof(int)} + 1 \]
  bytes when converting values in base 2 (binary).
- **radix**: The base to use when converting the number.

Library:
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `utoa()` function converts the unsigned binary integer *value* into the equivalent string in base *radix* notation, storing the result in the character array pointed to by *buffer*. A null character is appended to the result.

Returns:
A pointer to the result.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int base;
    char buffer[18];

    for( base = 2; base <= 16; base = base + 2 )
        printf( "%2d %s\n", base,
            utoa( (unsigned) 12765, buffer, base ) );
    return EXIT_SUCCESS;
}
```

produces the output:

```
2 11000111011101
4 3013131
6 135033
8 30735
10 12765
12 7479
14 491b
16 31dd
```

Classification:

QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>
See also:

`atoi()`, `atol()`, `itoa()`, `ltoa()`, `sscanf()`, `strtol()`, `strtoul()`, `ultoa()`
va_arg()

Get the next item in a list of variable arguments

Synopsis:

```c
#include <stdarg.h>

type va_arg( va_list param, type );
```

Arguments:

- `param` The `va_list` object that you initialized with the `va_start()` macro.
- `type` The type of the next argument.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

You can use the `va_arg()` macro to get the next argument in a list of variable arguments.

```
CAUTION: Take special care when using varargs on some platforms; see “Varargs and coercion,” below.
```

You must use `va_arg()` with the associated macros `va_copy()`, `va_start()` and `va_end()`. A sequence such as:

```c
void example( char *dst, ... )
{
    va_list curr_arg;
    int next_arg;

    va_start( curr_arg, dst );
    next_arg = va_arg( curr_arg, int );
```
causes next_arg to be assigned the value of the next variable argument. The argument type (which is int in the example) is the type of the argument originally passed to the function.

The last argument before the ellipsis (...) has to be an int or a type that doesn’t change in size if cast to an int. If the argument is promoted, the ANSI/ISO standard says the behavior is undefined, and so depends on the compiler and the library.

You must execute the macro va_start() first, in order to initialize the variable curr_arg properly, and execute the macro va_end() after getting all the arguments.

The data item curr_arg is of type va_list that contains the information to permit successive acquisitions of the arguments.

The following functions use a “varargs” list:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>verr()</td>
<td>vscanf()</td>
<td>vsyslog()</td>
</tr>
<tr>
<td>verrx()</td>
<td>vslogf()</td>
<td>vwarn()</td>
</tr>
<tr>
<td>vfprintf()</td>
<td>vsnprintf()</td>
<td>vwarnx()</td>
</tr>
<tr>
<td>vfscanf()</td>
<td>vprintf()</td>
<td>vwprintf()</td>
</tr>
<tr>
<td>vwprintf()</td>
<td>vsprintf()</td>
<td>vslogf()</td>
</tr>
<tr>
<td>vfwprintf()</td>
<td>vsnprintf()</td>
<td>vsprintf()</td>
</tr>
<tr>
<td>vfsscanf()</td>
<td>vfprintf()</td>
<td>vsscanf()</td>
</tr>
<tr>
<td>vfwscanf()</td>
<td>vscanf()</td>
<td>vwscanf()</td>
</tr>
</tbody>
</table>

Varargs and coercion

On some platforms, such as PowerPC, the va_list type is an array; on other platforms, such as x86, it isn’t. This can lead to problems.

Consider the following example. It seems correct, but on PowerPC platforms, it doesn’t print 2:

```c
#include <stdio.h>
#include <stdarg.h>

void handle_foo(char *fmt, va_list *pva) {
  printf("%d\n", va_arg(*pva, int));
}

void vfoo(char *fmt, va_list va) {
```
```c
void foo(char *fmt, ...) {
    va_list va;
    va_start(va, fmt);
    vfoo(fmt, va);
    va_end(va);
}

int main() {
    foo("", 2);
    return 0;
}
```

The C standard says that prototypes such as `vfoo()` have the array type silently coerced to be a pointer to a base type. This makes things work when you pass an array object to the function. An array-typed expression is converted to a pointer to the first element when used in an rvalue context, so the coercion in the function makes everybody happy.

The problem occurs when you then pass the address of the `va_list` parameter to another function. The function expects a pointer to the array, but what it really gets is a pointer to a pointer (because of the original conversion). If you use the `va_list` type in the second function, you won’t get the right data.

Here’s the example modified so that it works in all cases:

```c
#include <stdio.h>
#include <stdarg.h>

void handle_foo(char *fmt, va_list *pva) {
    printf("%d\n", va_arg(*pva, int));
}

void vfoo(char *fmt, va_list va) {
    va_list temp;
    va_copy(temp, va);
    handle_foo(fmt, &temp);
    va_end(temp);
}

void foo(char *fmt, ...) {
    handle_foo(fmt, &va);
}
```
va_list va;
va_start(va, fmt);
vfoo(fmt, va);
va_end(va);
}

int main()
{
    foo("", 2);
    return 0;
}

Using _va_copy() “undoes” the coercion that happens in the parameter list, so that _handle_foo() gets the proper data.

Returns:
The value of the next variable argument, according to type passed as the second parameter.

Examples:
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

static void test_fn( const char *msg,
    const char *types,
    ... );

int main( void )
{
    printf( "VA...TEST\n" );
test_fn( "PARAMETERS: 1, \"abc\", 546",
   "isi", 1, "abc", 546 );
test_fn( "PARAMETERS: \"def\", 789",
    "si", "def", 789 );
    return EXIT_SUCCESS;
}

static void test_fn( 
    const char *msg, /* message to be printed */
    const char *types, /* parameter types (i,s) */
    ... ) /* variable arguments */
{
    va_list argument;
    int  arg_int;
    char *arg_string;
    const char *types_ptr;
types_ptr = types;
printf( "\n%s -- %s\n", msg, types );
va_start( argument, types );
while( *types_ptr != '\0' ) {
    if (*types_ptr == 'i') {
        arg_int = va_arg( argument, int );
        printf( "integer: %d\n", arg_int );
    } else if (*types_ptr == 's') {
        arg_string = va_arg( argument, char * );
        printf( "string: %s\n", arg_string );
    }
    ++types_ptr;
} va_end( argument );

produces the output:

VA...TEST
PARAMETERS: 1, "abc", 546 -- isi
integer: 1
string: abc
integer: 546

PARAMETERS: "def", 789 -- si
string: def
integer: 789

Classification:

POSIX 1003.1

Safety

Cancellation point    No
Interrupt handler    Yes
Signal handler       Yes
Thread               Yes
Caveats:

\texttt{va\_arg()} is a macro.

See also:

\texttt{va\_copy()}, \texttt{va\_end()}, \texttt{va\_start()}
Synopsis:

```c
#include <stdarg.h>

void va_copy( va_list d,
              va_list s );
```

Arguments:

- `d` A `va_list` object into which you want to copy the list.
- `s` The `va_list` object that you initialized with the `va_start()` macro and that you want to copy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `va_copy()` macro creates a copy of a list of variable arguments.

You can use the `va_copy()` macro with the associated macros `va_arg()`, `va_start()`, and `va_end()`, especially to avoid problems on some platforms. For more information, see “Varargs and coercion” in the documentation for `va_arg()`.

Examples:

See `va_arg()`.

Classification:

POSIX 1003.1
**va_copy()**

<table>
<thead>
<tr>
<th><strong>Safety</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

va_copy() is a macro.

**See also:**

va_arg(), va_end(), va_start()
va_end()  
Finish getting items from a variable argument list

Synopsis:

```c
#include <stdarg.h>

void va_end( va_list param );
```

Arguments:

- `param`: The `va_list` object that you initialized with the `va_start()` macro.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

Use the `va_end()` macro to complete the acquisition of arguments from a list of variable arguments. You must use it with the associated macros `va_copy()`, `va_start()`, and `va_arg()`. For more information, see `va_arg()`.

Classification:

- POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

va_end() is a macro.

See also:

va_arg(), va_copy(), va_start()
va_start()

Start getting items from a variable argument list

Synopsis:

#include <stdarg.h>

void va_start ( va_list param, previous );

Arguments:

param A va_list object that the “varargs” macros can use to locate the arguments.

previous The argument that immediately precedes the "..." notation in the original function definition.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

Use the va_start() macro to start the acquisition of arguments from a list of variable arguments.

You must use the va_start() macro with the associated macros va_arg(), va_copy(), and va_end(). For each call to va_start(), you must have a matching call to va_end(). For more information, see va_arg().

Examples:

See va_arg().

Classification:

POSIX 1003.1
va_start() is a macro.

See also:
va_arg(), va_copy(), va_end()
valloc()

Allocate a heap block aligned on a page boundary

Synopsis:

```c
#include <stdarg.h>

void * valloc( size_t size);
```

Arguments:

- `size` The size of the block to allocate, in bytes.

Library:

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `valloc()` function allocates a heap block that’s aligned on a page boundary. It’s equivalent to:

```c
memalign( sysconf( _SC_PAGESIZE ), size);
```

Returns:

See `memalign()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`memalign()`, `sysconf()`
**verr(), verrx()**

Display a formatted error message, and then exit (varargs)

**Synopsis:**

```c
#include <err.h>

void verr( int eval, const char *fmt, va_list args);

void verrx( int eval, const char *fmt, va_list args);
```

**Arguments:**

- `eval` The value to use as the exit code of the process.
- `fmt` NULL, or a `printf()`-style string used to format the message.
- `args` A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

**Library:**

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `err()` and `warn()` family of functions display a formatted error message on `stderr`. For a comparison of the members of this family, see `err()`.

The `verr()` function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the `fmt` argument isn’t NULL
- the string associated with the current value of `errno`
verr(), verrx()

- a newline character.

The verrx() function produces a similar message, except that it doesn’t include the string associated with errno. The message consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, if the fmt argument isn’t NULL
- a newline character.

The verr() and verrx() functions don’t return, but exit with the value of the argument eval.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

err(), errx(), stderr, strerror(), vwarn(), vwarnx(), warn(), warnx()
vfork()

Spawn a new process and block the parent

Synopsis:

```c
#include <process.h>

pid_t vfork( void );
```

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

This function spawns a new process and blocks the parent until the child process calls `execve()` or exits (by calling `_exit()` or abnormally).

Returns:

A value of zero to the child process, and (later) the child’s process ID in the parent. If an error occurs, no child process is created, and the function returns -1 and sets `errno`.

Errors:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The system-imposed limit on the total number of processes under execution would be exceeded. This limit is determined when the system is generated.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>There isn’t enough memory for the new process.</td>
</tr>
</tbody>
</table>

Classification:

- POSIX 1003.1 XSI

Safety

- Cancellation point: No

continued . . .
Caveats:

To avoid a possible deadlock situation, processes that are children in the middle of a vfork() are never sent SIGTTOU or SIGTTIN signals; rather, output or ioctl commands are allowed and input attempts result in an EOF indication.

See also:

execve(), _exit(), fork(), ioctl(), sigaction(), wait()
Write formatted output to a file (varargs)

**Synopsis:**
```c
#include <stdio.h>
#include <stdarg.h>

int vfprintf( FILE* fp,
              const char* format,
              va_list arg );
```

**Arguments:**
- `fp` The stream to which you want to send the output.
- `format` A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.
- `arg` A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

**Library:**
- libc
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `vfprintf()` function writes output to the file pointed to by `fp`, under control of the argument `format`.

The `vfprintf()` function is a “varargs” version of `fprintf()`.

**Returns:**
The number of characters written, or a negative value if an output error occurred (`errno` is set).
Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

FILE *LogFile;

/* a general error routine */

void errmsg( const char *format, ... )
{
    va_list arglist;

    fprintf( stderr, "Error: " );
    va_start( arglist, format );
    vfprintf( stderr, format, arglist );
    va_end( arglist );
    if( LogFile != NULL ) {
        fprintf( LogFile, "Error: " );
        va_start( arglist, format );
        vfprintf( LogFile, format, arglist );
        va_end( arglist );
    }
}

int main( void )
{
    LogFile = fopen( "error.log", "w" );
    errmsg( "%s %d %s", "Failed", 100, "times" );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`errno`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`,
`va_start()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, `vsprintf()`, `vswprintf()`,
`vwprintf()`, `wprintf()`
vfscanf() — Scan input from a file (varargs)

Synopsis:

```c
#include <stdio.h>
#include <stdarg.h>

int vfscanf( FILE *fp,
             const char *format,
             va_list arg );
```

Arguments:

- **fp**  
The stream that you want to read from.
- **format**  
  A string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.
- **arg**  
  A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vfscanf()` function scans input from the file designated by `fp`, under control of the argument `format`.

The `vfscanf()` function is a “varargs” version of `fscanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is stopped by reaching the end of the input stream before storing any values.
Errors:

If an error occurs, `errno` indicates the type of error.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void ffind( FILE *fp, char *format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    vfscanf( fp, format, arglist );
    va_end( arglist );
}

int main( void )
{
    int day, year;
    char weekday[10], month[12];

    ffind( stdin,
        "%s %s %d %d",
        weekday, month, &day, &year );
    printf( "%s, %s %d, %d
",
        weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
vfscanf() © 2007, QNX Software Systems GmbH & Co. KG.

See also:

errno, fscanf(), fwscanf(), scanf(), sscanf(), swscanf(), va_start(),
vfwscanf(), vscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()
vfwprintf()

Write formatted wide-character output to a file (varargs)

Synopsis:

```c
#include <wchar.h>
#include <stdarg.h>

int vfwprintf( FILE * fp,
               const wchar_t * format,
               va_list arg );
```

Arguments:

- `fp` The stream to which you want to send the output.
- `format` A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.
- `arg` A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- `libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `vfwprintf()` function writes output to the file pointed to by `fp`, under control of the argument `format`.

The `vfwprintf()` function is the wide-character version of `vfprintf()`, and is a “varargs” version of `fwprintf()`.

Returns:

The number of wide characters written, excluding the terminating `NUL`, or a negative number if an error occurred (`errno` is set).
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`errno`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`, `va_start()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, `vsprintf()`, `vswprintf()`, `vwprintf()`, `wprintf()`
vfwscanf()  
Scan input from a file (varargs)

Synopsis:

```c
#include <wchar.h>
#include <stdarg.h>

int vfwscanf( FILE * fp,
               const wchar_t * format,
               va_list arg );
```

Arguments:

- **fp**  
The stream that you want to read from.

- **format**  
A wide-character string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

- **arg**  
A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- libc  
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vfwscanf()` function scans input from the file designated by `fp`, under control of the argument `format`.

The `vfwscanf()` function is the wide-character version of `vfscanf()`, and is a “varargs” version of `fwscanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.
**vfwscanf()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `fscanf()`, `fwscanf()`, `scanf()`, `sscanf()`, `swscanf()`, `va_start()`, `vfscanf()`, `vscanf()`, `vsscanf()`, `vswscanf()`, `vwscanf()`, `wscanf()`
vprintf()

Write formatted output to standard output (varargs)

Synopsis:

```c
#include <stdio.h>
#include <stdarg.h>

int vprintf( const char* format,
              va_list arg );
```

Arguments:

- **format**: A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.
- **arg**: A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vprintf()` function writes output to the file `stdout`, under control of the argument `format`.

The `vprintf()` function is a “varargs” version of `printf()`.

Returns:

The number of characters written, or a negative value if an output error occurred (`errno` is set).

Examples:

Use `vprintf()` in a general error message routine:

```c
#include <stdio.h>
#include <stdlib.h>
```
#include <stdarg.h>

void errmsg( const char* format, ... )
{
    va_list arglist;

    printf( "Error: " );
    va_start( arglist, format );
    vprintf( format, arglist );
    va_end( arglist );
}

int main( void )
{
    errmsg( "%s %d %s", "Failed", 100, "times" );
    return EXIT_SUCCESS;
}

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

errno, fprintf(), fwprintf(), printf(), snprintf(), sprintf(), swprintf(), va_start(), vfprintf(), vfwprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
vscanf()  
Scan input from standard input (varargs)

Synopsis:

```c
#include <stdio.h>
#include <stdarg.h>

int vscanf( const char * format, 
            va_list args );
```

Arguments:

- **format**  
  A string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

- **args**  
  A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- **libc**  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vscanf()` function scans input from `stdin`, under control of the argument `format`. For information about the `format` string, see the description of `scanf()`.

The `vscanf()` function is a “varargs” version of `scanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF when the scanning is stopped by reaching the end of the input stream before storing any values.
Errors:
If an error occurs, \textit{errno} indicates the type of error.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void find( char *format, ... )
{
    va_list arglist;
    va_start( arglist, format );
    vscanf( format, arglist );
    va_end( arglist );
}

int main( void )
{
    int day, year;
    char weekday[10], month[12];

    ffend( "%s %s %d %d",
        weekday, month, &day, &year );
    printf( "\n%s, %s %d, %d\n",
        weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:
ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Description</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
See also:

`errno`, `fscanf()`, `fwscanf()`, `scanf()`, `sscanf()`, `swscanf()`, `va_start()`, `vfscanf()`, `vfwscanf()`, `vsscanf()`, `vswscanf()`, `vvsscanf()`, `vwscanf()`, `wscanf()`
vslogf() © 2007, QNX Software Systems GmbH & Co. KG.

Send a message to the system logger (varargs)

Synopsis:

```c
#include <stdio.h>
#include <sys/slog.h>

int vslogf(int opcode,
            int severity,
            const char *fmt,
            va_list arg);
```

Arguments:

- **opcode**: A combination of a major and minor code. Create the opcode using the `_SLOG_SETCODE(major, minor)` macro that’s defined in `<sys/slog.h>`. The major and minor codes are defined in `<sys/slogcodes.h>`.

- **severity**: The severity of the log message; see “Severity levels,” in the documentation for `slogf()`.

- **fmt**: A standard `printf()` string.

- **arg**: A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- **libc**: Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).
Description:

The `slog*()` functions send log messages to the system logger, `slogger`. To send formatted messages, use `vslogf()`. If you have programs that scan log files for specified codes, you can use `slogb()` or `slogi()` to send a block of structures or `int`’s, respectively.

This function is a “varargs” version of `slogf()`.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`slogb(), slogi(), slogf()`  
`slogger, sloginfo` in the Utilities Reference
vsnprintf() © 2007, QNX Software Systems GmbH & Co. KG.

Write formatted output to a character array, up to a maximum number of characters (varargs)

Synopsis:
```c
#include <stdarg.h>
#include <stdio.h>

int vsnprintf( char* buf,
               size_t count,
               const char* format,
               va_list arg );
```

Arguments:

- `buf` A pointer to the buffer where you want to function to store the formatted string.
- `count` The maximum number of characters to store in the buffer, including a terminating null character.
- `format` A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.
- `arg` A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vsnprintf()` function formats data under control of the `format` control string and stores the result in `buf`. The maximum number of characters to store, including a terminating null character, is specified by `count`.

The `vsnprintf()` function is a “varargs” version of `snprintf()`.
vsnprintf()

Returns:

The number of characters that would have been written into the array, not counting the terminating null character, had count been large enough. It does this even if count is zero; in this case buf can be NULL.

If an error occurred, vsnprintf() returns a negative value and sets errno.

Examples:

Use vsnprintf() in a general error message routine:

```c
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

char msgbuf[80];

char *fmtmsg( char *format, ... )
{
    va_list arglist;
    va_start( arglist, format );
    strcpy( msgbuf, "Error: " );
    vsnprintf( &msgbuf[7], 80-7, format, arglist );
    va_end( arglist );
    return( msgbuf );
}

int main( void )
{
    char *msg;
    msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
    printf( "%s\n", msg );
    return 0;
}
```

Classification:

ANSI, POSIX 1003.1
vsnprintf()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

It’s safe to call `vsnprintf()` in a signal handler if the data isn’t floating point.

See also:

`errno`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`, `va_start()`, `vfprintf()`, `vfwprintf()`, `vprintf()`, `vsnprintf()`, `vswprintf()`, `vwprintf()`, `wprintf()`
**Synopsis:**

```c
#include <stdio.h>
#include <stdarg.h>

int vsprintf( char* buf,
              const char* format,
              va_list arg );
```

**Arguments:**

- **buf** A pointer to the buffer where you want to function to store the formatted string.
- **format** A string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.
- **arg** A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

**Library:**

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `vsprintf()` function formats data under control of the `format` control string, and writes the result to `buf`.

The `vsprintf()` function is a “varargs” version of `sprintf()`.

**Returns:**

The number of characters written, or a negative value if an output error occurred (`errno` is set).
Examples:

Use `vsprintf()` in a general error message routine:

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>

char msgbuf[80];

char *fmtmsg( char *format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    strcpy( msgbuf, "Error: ");
    vsprintf( &msgbuf[7], format, arglist );
    va_end( arglist );
    return( msgbuf );
}

int main( void )
{
    char *msg;

    msg = fmtmsg( "%s %d %s", "Failed", 100, "times" );
    printf( "%s\n", msg );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

It’s safe to call `vsprintf()` in a signal handler if the data isn’t floating point.

See also:

- `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`, `va_start()`, `vfprintf()`, `vfwprintf()`, `vprintf()`, `vsnprintf()`, `vswprintf()`, `vwprintf()`, `wprintf()`
**vsscanf()**  
© 2007, QNX Software Systems GmbH & Co. KG.

Scan input from a string (varargs)

**Synopsis:**
```c
#include <stdio.h>
#include <stdarg.h>

int vsscanf( const char* in_string, 
             const char* format, 
             va_list arg );
```

**Arguments:**
- **in_string** The string that you want to read from.
- **format** A string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.
- **arg** A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

**Library:**
- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `vsscanf()` function scans input from the string designated by `in_string`, under control of the argument `format`.

The `vsscanf()` function is a “varargs” version of `sscanf()`.

**Returns:**

The number of input arguments for which values were successfully scanned and stored is returned, or EOF when the scanning is terminated by reaching the end of the input string.
**Examples:**

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

void sfind( char* string, char* format, ... )
{
    va_list arglist;

    va_start( arglist, format );
    vsscanf( string, format, arglist );
    va_end( arglist );
}

int main( void )
{
    int day, year;
    char weekday[10], month[12];

    sfind( "Monday June 28 1999",
    "%s %s %d %d",
    weekday, month, &day, &year );
    printf( "%s, %s %d, %d
",
    weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

It’s safe to call `vsscanf()` in a signal handler if the data isn’t floating point.

See also:

`fscanf()`, `fwscanf()`, `scanf()`, `sscanf()`, `swscanf()`, `va_start()`, `vfscanf()`,
`vfwscanf()`, `vscanf()`, `vsscanf()`, `vwscanf()`, `wscanf()`
vswprintf()

Write wide-character formatted output to a buffer (varargs)

Synopsis:

```c
#include <wchar.h>
#include <stdarg.h>

int vswprintf( wchar_t *buf,
               size_t n,
               const wchar_t *format,
               va_list arg);
```

Arguments:

- **buf** A pointer to the buffer where you want to function to store the formatted string.
- **n** The maximum number of wide characters to store in the buffer, including a terminating null character.
- **format** A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.
- **arg** A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vswprintf()` function formats data under control of the `format` control string, and writes the result to `buf`.

The `vswprintf()` function is the wide-character version of `vprintf()`, and is a “varargs” version of `swprintf()`.
Returns:

The number of wide characters written, excluding the terminating NUL, or a negative number if an error occurred (errno is set).

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

It’s safe to call vswprintf() in a signal handler if the data isn’t floating point.

See also:

fprintf(), fwprintf(), printf(), snprintf(), sprintf(), swprintf(), va_start(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vwprintf(), wprintf()
vswscanf()

Scan input from a wide-character string (varargs)

Synopsis:

```c
#include <wchar.h>
#include <stdarg.h>

int vswscanf( const wchar_t * ws,
               const wchar_t * format,
               va_list arg );
```

Arguments:

- `ws` The wide-character string that you want to read from.
- `format` A wide-character string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.
- `arg` A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vswscanf()` function scans input from the string designated by `ws`, under control of the argument `format`.

The `vswscanf()` function is the wide-character version of `vsscanf()`, and is a “varargs” version of `swscanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored is returned, or EOF when the scanning is terminated by reaching the end of the input string.
vswscanf()

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

It’s safe to call `vswscanf()` in a signal handler if the data isn’t floating point.

See also:

`fscanf()`, `fwscanf()`, `scanf()`, `sscanf()`, `swscanf()`, `va_start()`, `vfscanf()`, `vfwscanf()`, `vscanf()`, `vsscanf()`, `vwscanf()`, `wscanf()`
vsyslog()
Control system log (varargs)

Synopsis:

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog( int priority,
        const char *message,
        va_list args);
```

Arguments:

- **priority**  The priority of the message; see “Message levels,” in the documentation for syslog().
- **message**  The message that you want to write. This message is identical to a printf()-format string, except that %m is replaced by the current error message (as denoted by the global variable errno). A trailing newline is added if none is present.
- **args**  A variable-argument list of the additional arguments, which you must have initialized with the va_start() macro.

Library:

```
libc
```

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The vsyslog() function writes message to the system message logger. The message is then written to the system console, log files, and logged-in users, or forwarded to other machines as appropriate. (See the syslogd command.)

This function is a “varargs” version of syslog().
**vsyslog()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`closelog(), openlog(), setlogmask(), syslog()`  
`logger, syslogd` in the *Utilities Reference*
Synopsis:

```c
#include <err.h>

void vwarn( const char *fmt,
            va_list args );

void vwarnx( const char *fmt,
             va_list args );
```

Arguments:

- **fmt**: NULL, or a `printf()`-style string used to format the message.
- **args**: A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

```c
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `err()` and `warn()` family of functions display a formatted error message on `stderr`. For a comparison of the members of this family, see `err()`.

The `vwarn()` function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the `fmt` argument isn’t NULL
- the string associated with the current value of `errno`
- a newline character.
vwarn(), vwarnx()

The vwarnx() function produces a similar message, except that it doesn’t include the string associated with errno. The message consists of:
- the last component of the program name, followed by a colon and a space
- the formatted message, if the fmt argument isn’t NULL
- a newline character.

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

err(), errx(), stderr, strerror(), verr(), verrx(), warn(), warnx()
vwprintf()

Write formatted output to standard output (varargs)

Synopsis:

```c
#include <wchar.h>
#include <stdarg.h>

int vwprintf( const wchar_t * format,
               va_list arg );
```

Arguments:

- `format`: A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

- `arg`: A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vwprintf()` function writes output to the file `stdout`, under control of the argument `format`.

The `vwprintf()` function is the wide-character version of `vprintf()`, and is a “varargs” version of `wprintf()`.

Returns:

The number of characters written, or a negative value if an output error occurred (`errno` is set).
**vwprintf()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

errno, fprintf(), fwprintf(), printf(), snprintf(), sprintf(), swprintf(), va_start(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), wprintf()
vwscanf()

Scan wide-character input from standard input (varargs)

Synopsis:

```c
#include <wchar.h>
#include <stdarg.h>

int vwscanf( const wchar_t * format,
             va_list arg );
```

Arguments:

- **format**  
  A wide-character string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

- **arg**  
  A variable-argument list of the additional arguments, which you must have initialized with the `va_start()` macro.

Library:

- **libc**  
  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `vwscanf()` function scans input from the file designated by `stdin`, under control of the argument `format`.

The `vwscanf()` function is the wide-character version of `vscanf()`, and is a “varargs” version of `wscanf()`.

Returns:

The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.
vwscanf() © 2007, QNX Software Systems GmbH & Co. KG.

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fscanf(), fwscanf(), scanf(), sscanf(), swscanf(), va_start(), vfscanf(), vfwscanf(), vscanf(), vsscanf(), vswscanf(), wscanf()
**wait()**

Wait for the status of a terminated child process

**Synopsis:**
```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait( int * stat_loc);
```

**Arguments:**

`stat_loc` NULL, or a pointer to a location where the function can store the terminating status of the child. For more information, see “Status macros,” below.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wait()` function suspends execution of the calling process until status information from one of its terminated child processes is available, or until the delivery of a signal whose action is either to terminate the process or execute a signal handler. If status information is available prior to the call to `wait()`, the return is immediate.

**Status macros**

If the `stat_loc` variable is non-NULL, the terminating status of the child process is in the location that it points to. The macros listed below, defined in `<sys/wait.h>`, extract information from `stat_loc`. The `stat_val` argument to these macros is the integer value pointed to by `stat_loc`.

POSIX defines the following macros:
WEXITSTATUS( *stat_val )

Evaluates to the low-order 8 bits of the termination status of the child process if the value of WIFEXITED( *stat_val ) is nonzero.

WIFCONTINUED( *stat_val )

Evaluates to a nonzero value if the status returned was from a child process that has continued from a job control stop.

WIFEXITED( *stat_val )

Evaluates to a nonzero value if the status returned was from a normally terminated child process.

WIFSIGNALED( *stat_val )

Evaluates to nonzero value if the child process terminated from reception of a signal that wasn’t caught.

WIFSTOPPED( *stat_val )

Evaluates to a nonzero value if the status returned is for a child process that’s stopped.

WSTOPSIG( *stat_val )

Evaluates to the number of the signal that caused the child process to stop if the value of WIFSTOPPED( *stat_val ) is nonzero.

WTERMSIG( *stat_val )

Evaluates to the number of the signal that terminated the child process if the value of WIFSIGNALED( *stat_val ) is nonzero.

This macro isn’t part of a POSIX standard:

WCOREDUMP( *stat_val )

Evaluates to a nonzero value if the child process left a core dump.

One of the macros WIFEXITED(*stat_loc) and WIFSIGNALED(*stat_loc) evaluates to a nonzero value.
The non-POSIX `waitid()` function gives even more status information than the above macros.

**Returns:**

The process ID of the terminating child process, or -1 if an error occurred or on delivery of a signal (`errno` is set to EINTR).

**Errors:**

- **ECHILD** The calling process has no existing unwaited-for child processes.
- **EINTR** The function was interrupted by a signal. The value of the location pointed to by `stat_loc` is undefined.

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`errno`, `spawn()`, `wait4()`, `waitid()`, `waitpid()`
Wait for any child process to change its state

Synopsis:

```c
#include <sys/wait.h>

pid_t wait3( int * stat_loc,
             int options,
             struct rusage * resource_usage );
```

Arguments:

- `stat_loc` NULL, or a pointer a location where the function can store the terminating status of the child process. For information about macros that extract information from this status, see “Status macros” in the documentation for `wait()`.

- `options` A combination of zero or more of the following flags:
  - `WCONTINUED` — return the status for any child that was stopped and has been continued.
  - `WEXITED` — wait for the process(es) to exit.
  - `WNOHANG` — return immediately if there are no children to wait for.
  - `WSTOPPED` — wait for and return the process status of any child that has stopped because it received a signal.
  - `WUNTRACED` — report the status of a stopped child process.

- `resource_usage` NULL, or a pointer to a `rusage` structure where the function can store information about resource usage. For information about this structure, see `getrusage()`.
Library:

```
libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
```

Description:

The `wait3()` function allows the calling thread to obtain status information for specified child processes.

The following call:

```
wait3( stat_loc, options, resource_usage );
```

is equivalent to the call:

```
waitpid( (pid_t)-1, stat_loc, options );
```

except that on successful completion, if the `resource_usage` argument to `wait3()` isn’t a null pointer, the `rusage` structure that the third argument points to is filled in for the child process identified by the return value.

It’s also equivalent to:

```
wait4( (pid_t)-1, stat_loc, options, resource_usage );
```

Returns:

If the status of a child process is available, a value equal to the process ID of the child process for which status is reported.

If a signal is delivered to the calling process, -1 and `errno` is set to EINTR.

Zero if `wait3()` is invoked with WNOHANG set in `options` and at least one child process is specified by `pid` for which status isn’t available, and status isn’t available for any process specified by `pid`.

Otherwise, `(pid_t)-1` and `errno` is set.
wait3()

Errors:

ECHILD The calling process has no existing unwaited-for child processes, or the set of processes specified by the argument pid can never be in the states specified by the argument options.

Classification:

Unix

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

New applications should use waitpid().

See also:

exit(), fork(), pause(), wait4(), waitid(), waitpid()
wait4()

Wait for one or more child process to change its state

Synopsis:

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait4( pid_t pid,
             int * stat_loc,
             int options,
             struct rusage * resource_usage );
```

Arguments:

- `pid` The set of child processes that you want to get status information for:
  - less than -1 — any child process whose process group ID is equal to the absolute value of `pid`.
  - -1 — any child process
  - 0 — any child process whose process group ID is equal to that of the calling process.
  - greater than 0 — the single child process with this ID.

- `stat_loc` NULL, or a pointer a location where the function can store the terminating status of the child process. For information about macros that extract information from this status, see “Status macros” in the documentation for `wait()`.

- `options` A combination of zero or more of the following flags:
  - WCONTINUED — return the status for any child that was stopped and has been continued.
  - WEXITED — wait for the process(es) to exit.
  - WNOHANG — return immediately if there are no children to wait for.
  - WNOWAIT — keep the process in a waitable state. This doesn’t affect the state of the process; the process may be waited for again after this call completion.
wait4()

- WSTOPPED — wait for and return the process status of any child that has stopped because it received a signal.
- WUNTRACED — report the status of a stopped child process.

resource_usage

NULL, or a pointer to a rusage structure where the function can store information about resource usage. For information about this structure, see getrusage().

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The wait4() function suspends execution of the calling process until status information from one of its terminated child processes is available, or until the delivery of a signal whose action is either to terminate the process or execute a signal handler. If status information is available prior to the call to wait4(), the return is immediate.

The wait4() function behaves the same as the wait() function when passed a pid argument of -1, and the options argument has a value of zero.

Only one of the WIFEXITED(stat_val) and WIFSIGNALED(stat_val) macros can evaluate to a nonzero value.

The following call:

```c
wait3( stat_loc, options, resource_usage );
```

is equivalent to the call:

```c
waitpid( (pid_t)-1, stat_loc, options );
```
wait4()

except that on successful completion, if the resource_usage argument to wait3() isn’t a NULL pointer, the rusage structure that the third argument points to is filled in for the child process identified by the return value.

It’s also equivalent to:

```
wait4( (pid_t)-1, stat_loc, options, resource_usage );
```

**Returns:**

If successful, wait4() returns the process id of the terminating child process. If wait4() was invoked with WNOHANG set in options, it has at least one child process specified by pid for which status is not available, and status is not available for any process specified by pid, a value of zero is returned. On delivery of a signal waitpid() returns -1, and errno is set to EINTR.

**Errors:**

ECHILD    The calling process has no existing unwaited-for child processes that meet the criteria set by pid.

EINVAL    The value of the options argument isn’t valid.

**Classification:**

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

September 10, 2007 C Library — T to Z 3703
See also:

exit(), fork(), pause(), wait(), wait3(), waitid(), waitpid()
waitid()

Wait for a child process to change state

Synopsis:

```c
#include <sys/wait.h>

int waitid( idtype_t idtype,
            id_t id,
            siginfo_t * infop,
            int options );
```

Arguments:

- **idtype**: Which children you want to wait for:
  - P_PID — the child with a process ID of (pid_t) id.
  - P_PGID — any child with a process group ID equal to (pid_t) id.
  - P_ALL — any child; id is ignored.

- **id**: The process or process group ID that you want to wait for, depending on the value of idtype.

- **infop**: A pointer to a siginfo_t structure, as defined in <sys/siginfo.h>, where the function can store the current state of the child; see below.

- **options**: A combination of zero or more of the following flags:
  - WCONTINUED — return the status for any child that was stopped and has been continued.
  - WEXITED — wait for the process(es) to exit.
  - WNOHANG — return immediately if there are no children to wait for.
  - WNOWAIT — keep the process in a waitable state. This doesn’t affect the state of the process; the process may be waited for again after this call completion.
  - WSTOPPED — wait for and return the process status of any child that has stopped because it received a signal.
  - WUNTRACED — report the status of a stopped child process.
waitid()

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The waitid() function suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by infop. If a child process changed state prior to the call to waitid(), waitid() returns immediately.

If waitid() returns because a child process was found that satisfied the conditions indicated by the arguments idtype and options, then the structure pointed to by infop is filled in by the system with the status of the process. The si_signo member is always SIGCHLD.

If idtype is P_ALL and options is WEXITED|WTRAPPED, waitid() is equivalent to wait().

Returns:

0 One of the children changed its state. If WNOHANG was used, 0 can be returned (indicating no error); however, no children may have changed state if info->si_pid is 0.

-1 An error occurred (errno is set).

Errors:

ECHILD The set of processes specified by idtype and id doesn’t contain any unwaited-for processes.

EFAULT The infop argument points to an illegal address.

EINTR The waitid() function was interrupted by a signal.

EINVAL An invalid value was specified for options, or idtype and id specify an invalid set of processes.
Classification:

POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`excl()`, `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`,
`execvpe()`, `exit()`, `fork()`, `pause()`, `sigaction()`, `signal()`, `wait()`, `wait3()`,
`wait4()`, `waitpid()`
waitpid()

Suspend the calling process

Synopsis:

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid( pid_t pid,
              int * stat_loc,
              int options );
```

Arguments:

`pid` The set of child processes that you want to get status information for:

- less than -1 — any child process whose process group ID is equal to the absolute value of `pid`.
- -1 — any child process
- 0 — any child process whose process group ID is equal to that of the calling process.
- greater than 0 — the single child process with this ID.

`stat_loc` NULL, or a pointer a location where the function can store the terminating status of the child process. For information about macros that extract information from this status, see “Status macros” in the documentation for `wait()`.

`options` A combination of zero or more of the following flags:

- WCONTINUED — return the status for any child that was stopped and has been continued.
- WEXITED — wait for the process(es) to exit.
- WNOHANG — return immediately if there are no children to wait for.
- WNOWAIT — keep the process in a waitable state. This doesn’t affect the state of the process; the process may be waited for again after this call completion.
waitpid()

- WSTOPPED — wait for and return the process status of any child that has stopped because it received a signal.
- WUNTRACED — report the status of a stopped child process.

Library:

libc

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `waitpid()` function suspends execution of the calling process until status information from one of its terminated child processes is available, or until the delivery of a signal whose action is either to terminate the process or execute a signal handler. If status information is available prior to the call to `waitpid()`, the return is immediate.

The `waitpid()` function behaves the same as `wait()` when passed a `pid` argument of -1, and the `options` argument has a value of zero.

Only one of the `WIFEXITED(stat_val)` and `WIFSIGNALED(stat_val)` macros can evaluate to a nonzero value.

Returns:

The process ID of the terminating child process on success. If `waitpid()` is invoked with `WNOHANG` set in `options`, it has at least one child process specified by `pid` for which status isn’t available, and status isn’t available for any process specified by `pid`, a value of zero is returned. On delivery of a signal, `waitpid()` returns -1, and `errno` is set to EINTR.
**waitpid()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Errors:**

ECHILD  The calling process has no existing unwaited-for child processes that meet the criteria set by *pid*.

EINTR   The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EINVAL  The value of the *options* argument isn’t valid.

**Classification:**

POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

spawn(), wait(), wait3(), wait4(), waitid()
Synopsis:

```c
#include <err.h>

void warn( const char* fmt, ...);
void warnx( const char* fmt, ...);
```

Arguments:

- `fmt` NULL, or a `printf()`-style string used to format the message.
  Additional arguments
  - As required by the format string.

Library:

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `err()` and `warn()` family of functions display a formatted error message on `stderr`. For a comparison of the members of this family, see `err()`.

The `warn()` function produces a message that consists of:

- the last component of the program name, followed by a colon and a space
- the formatted message, followed by a colon and a space, if the `fmt` argument isn’t NULL
- the string associated with the current value of `errno`
- a newline character.

The `warnx()` function produces a similar message, except that it doesn’t include the string associated with `errno`. The message consists of:
warn(), warnx()

- the last component of the program name, followed by a colon and a space
- the formatted message, if the fmt argument isn’t NULL
- a newline character.

Examples:

Warn of an error:

```c
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
    warnx("%s: %s: trying the block device",
       raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
    warn("%s", block_device);
```

Classification:

Unix

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

err(), errx(), stderr, strerror(), err(), verrx(), vwarn(), vwarnx()
**Synopsis:**

```c
#include <wchar.h>

size_t wcrtomb( char * s, 
    wchar_t wc, 
    mbstate_t * ps );
```

**Arguments:**

- `s` NULL, or a pointer to a location where the function can store the multibyte character.
- `wc` The wide character that you want to convert.
- `ps` An internal pointer that lets `wcrtomb()` be a restartable version of `wctomb()`; if `ps` is NULL, `wcrtomb()` uses its own internal variable.
  
  You can call `mbsinit()` to determine the status of this variable.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wcrtomb()` function determines the number of bytes needed to represent the wide character `wc` as a multibyte character and stores the multibyte character in the location pointed to by `s`, to a maximum of `MB_CUR_MAX` bytes.

This function is affected by `LC_CTYPE`.

**Returns:**

The number of bytes stored, or `(size_t) -1` if the variable `wc` is an invalid wide-character code.
**Errors:**

- **EILSEQ** Invalid wide-character code.
- **EINVAL** The variable *ps* points to an invalid conversion state.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**Synopsis:**

```c
#include <wchar.h>

wchar_t * wcscat( wchar_t * ws1,
                  const wchar_t * ws2 );
```

**Arguments:**

`ws1, ws2`  The wide-character strings that you want to concatenate.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wcscat()` function appends a copy of the string pointed to by `ws2`, including the terminating NUL wide character, to the end of the string pointed to by `ws1`. The first wide character of `ws2` overwrites the NUL wide character at the end of `ws1`.

**Returns:**

The same pointer as `ws1`.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: Yes

*continued.*
**wcscat()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**wcschr()**

Find the first occurrence of a wide character in a string

**Synopsis:**

```c
#include <wchar.h>

wchar_t * wcschr( const wchar_t * ws, wchar_t wc );
```

**Arguments:**

- `ws` The wide-character string that you want to search.
- `wc` The wide character that you’re looking for.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wcschr()` function finds the first occurrence of `wc` in the string pointed to by `ws`. The terminating NUL character is considered to be part of the string.

**Returns:**

A pointer to the located wide character, or NULL if `wc` doesn’t occur in the string.

**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

*continued.*
**wcschr()**

© 2007, QNX Software Systems GmbH & Co. KG.

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `wcscespn()`, `wcspbrk()`, `wcsrchr()`, `wcsspnn()`, `wcsstr()`, `wcstok()`
Synopsis:

```c
#include <wchar.h>

int wcscmp( const wchar_t * ws1,
             const wchar_t * ws2 );
```

Arguments:

- `ws1`, `ws2`  The wide-character strings that you want to compare.

Library:

- libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcscmp()` function compares the wide-character strings pointed to by `ws1` and `ws2`.

Returns:

- `< 0`  `ws1` is less than `ws2`.
- `0`  `ws1` is equal to `ws2`.
- `> 0`  `ws1` is greater than `ws2`.

Classification:

- ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
\textbf{wcscmp()}

\begin{center}
\begin{tabular}{|l|c|}
\hline
\textbf{Safety} & \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}
\end{center}

\textbf{See also:}
\begin{itemize}
\item strcasecmp(), strcmp(), strcmpi(), strcoll(), stricmp(), strncasecmp(),
\item strncmp(), strnicmp(), wcscoll(), wcscmp()
\end{itemize}
wcscoll()

Compare two wide-character strings, using the locale’s collating sequence

Synopsis:

```c
#include <wchar.h>

int wcscoll( const wchar_t * ws1, const wchar_t * ws2 );
```

Arguments:

- `ws1, ws2`: The wide-character strings that you want to compare.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcscoll()` function compares the wide-character strings pointed to by `ws1` and `ws2`, using the LC_COLLATE collating sequence selected by the `setlocale()` function.

Returns:

- `< 0`: `ws1` is less than `ws2`.
- `0`: `ws1` is equal to `ws2`.
- `> 0`: `ws1` is greater than `ws2`.

Classification:

- ANSI, POSIX 1003.1

Safety

- Cancellation point: No

continued…
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

### See also:

setlocale(), strcasecmp(), strcmp(), strcmpi(), strcoll(), stricmp(), strncasecmp(), strncmp(), strnicmp(), wcscmp(), wcsncmp()
Synopsis:

```c
#include <wchar.h>

wchar_t * wcscpy( wchar_t * ws1,
                    const char * ws2 );
```

Arguments:

- `ws1` A pointer to where you want to copy the string.
- `ws2` The wide-character string that you want to copy.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcscpy()` function copies the string pointed to by `ws2`, including the terminating NUL wide character, into the array pointed to by `ws1`.

This function isn’t guaranteed to work properly for copying overlapping strings; use `wmemmove()` instead.

Returns:

The same pointer as `ws1`.

Classification:

ANSI, POSIX 1003.1

Safety

Cancellation point No

...continued
wcscpy()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memmove(), strcpy(), strdup(), strncpy(), wcsncpy(), wmemmove()
Count the wide characters at the beginning of a string that aren’t in a given character set

**Synopsis:**

```c
#include <wchar.h>

size_t wcscspn( const wchar_t * ws1, const wchar_t * ws2 );
```

**Arguments:**

- `ws1` The wide-character string that you want to search.
- `ws2` The set of wide characters you want to look for.

**Library:**

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `strspn()` function returns the length of the initial segment of the string pointed to by `ws1` consisting entirely of wide characters not from the string pointed to by `ws2`. The terminating NUL isn’t considered to be part of `ws2`.

**Returns:**

The length of the segment.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: Yes

*continued...*
See also:

memchr(), strchr(), strcspn(), strpbrk(), strrchr(), strspn(), strstr(), strtok(), strtok_r(), wcschr(), wcspbrk(),wcsrchr(), wcsspn(), wcsstr(), wcstok()
wcsftime()

Format the time into a wide-character string

Synopsis:

```c
#include <wchar.h>

size_t wcsftime( wchar_t * wcs,  
                 size_t maxsize,  
                 const wchar_t * format,  
                 const struct tm * timeptr );
```

Arguments:

- `wcs` A pointer to a buffer where the function can store the formatted time.
- `maxsize` The maximum size of the buffer.
- `format` The format that you want to use for the time; see “Formats,” in the description of `strftime()`.
- `timeptr` A pointer to a `tm` structure that contains the time that you want to format.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsftime()` function is similar to `strftime()`, except that `wcsftime()` works with wide characters.

Returns:

The number of wide characters placed into the array, not including the terminating null character, or 0 if the number of wide characters exceeds `maxsize` (in this case, the string contents are indeterminate).

If an error occurs, `errno` is set.
wcsftime()

Classification:

ANSI, POSIX 1003.1

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

asctime(), asctime_r(), ctime(), ctime_r(), strftime(), strftime(), tm, tzset()
wcslen()  
Compute the length of a wide-character string

Synopsis:

```c
#include <wchar.h>

size_t wcslen( const wchar_t * ws );
```

Arguments:

- `ws`  The wide-character string whose length you want to calculate.

Library:

- libc

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The `wcslen()` function counts the wide characters in the string pointed to by `ws`.

Returns:

The number of wide characters, not counting the terminating NUL.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

\textit{strlen()}

wcsncat()

Concatenate two wide-character strings, up to a maximum length

Synopsis:

```c
#include <wchar.h>

wchar_t * wcsncat( wchar_t * ws1,
                   const wchar_t * ws2,
                   size_t n );
```

Arguments:

- `ws1, ws2`: The wide-character strings that you want to concatenate.
- `n`: The maximum number of wide characters that you want to add from the `ws2` string.

Library:

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsncat()` function appends a copy of the string pointed to by `ws2`, including the terminating NUL wide character, to the end of the string pointed to by `ws1`. The first character of `ws2` overwrites the NUL wide character at the end of `ws1`. The function writes no more than `n` wide characters from `ws2` and appends a NUL wide character to the result.

Returns:

The same pointer as `ws1`.

Classification:

- ANSI, POSIX 1003.1
### wcsncat()

**Safety**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```
#include <wchar.h>

int wcsncmp( const wchar_t * ws1, const wchar_t * ws2, size_t n );
```

Arguments:

- `ws1, ws2` The wide-character strings that you want to compare.
- `n` The maximum number of wide characters that you want to compare.

Library:

`libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsncmp()` function compares up `n` wide characters from the strings pointed to by `ws1` and `ws2`.

Returns:

- `< 0` `ws1` is less than `ws2`.
- `0` `ws1` is equal to `ws2`.
- `> 0` `ws1` is greater than `ws2`.

Classification:

ANSI, POSIX 1003.1
wcsncmp() © 2007, QNX Software Systems GmbH & Co. KG.

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

strcasemyp(), strcmp(), strempy(), strcoll(), stricmp(), strncasemyp(), strncmp(), strnicmp(), wcscmp(), wcscoll()
Synopsis:

```
#include <wchar.h>

wchar_t * wcsncpy( wchar_t * ws1,
                   const char * ws2,
                   size_t n );
```

Arguments:

- **ws1**: A pointer to where you want to copy the wide-character string.
- **ws2**: The wide-character string that you want to copy.
- **n**: The maximum number of wide characters that you want to copy.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsncpy()` function copies the string pointed to by `ws2`, including the terminating NUL wide character, into the array pointed to by `ws1`, to a maximum of `n` wide characters. It adds NUL characters if `ws2` has fewer than `n` characters but doesn’t add a NUL if `ws2` has more.

This function isn’t guaranteed to work properly for copying overlapping strings; use `wmemmove()` instead.

Returns:

The same pointer as `ws1`. 
## wcsncpy()

### Classification:

ANSI, POSIX 1003.1

### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`memmove()`, `strcpy()`, `strdup()`, `strncpy()`, `wcsncpy()`, `wmemmove()`
Synopsis:

```c
#include <wchar.h>

wchar_t * wcspbrk( const wchar_t * ws1, const wchar_t * ws2 );
```

Arguments:

- `ws1` The wide-character string that you want to search.
- `ws2` The set of wide characters you want to look for.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcspbrk()` function locates the first occurrence in the string pointed to by `ws1` of any wide character from the string pointed to by `ws2`.

Returns:

A pointer to the located character, or NULL if no character from `ws2` occurs in `ws1`.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

continued...
Safety

<table>
<thead>
<tr>
<th>Signal handler</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`,
`strtok()`, `strtok_r()`, `wcschr()`, `wcscespn()`, `wcsecchr()`, `wcsstr()`,
`wcsstr()`, `wcestok()`
**wcsrchr()**

*Find the last occurrence of a wide character in a string*

**Synopsis:**
```
#include <wchar.h>

wchar_t * wcsrchr( const wchar_t * ws,
                   wchar_t wc );
```

**Arguments:**

- `ws` The wide-character string that you want to search.
- `wc` The wide character that you’re looking for.

**Library:**
```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wcsrchr()` function finds the last occurrence of `wc` in the string pointed to by `ws`. The terminating NUL character is considered to be part of the string.

**Returns:**

A pointer to the located wide character, or NULL if `wc` doesn’t occur in the string.

**Classification:**

- ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
</tbody>
</table>

*continued...*
See also:

memchr(), strchr(), strcspn(), strpbrk(), strrchr(), strspn(), strstr(), strtok(), strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcscspn(), wcsstr(), wcstok()
wcsrtombs()

Convert a wide-character string into a multibyte character string (restartable)

Synopsis:

```c
#include <wchar.h>

size_t wcsrtombs( char * dst,
                  const wchar_t ** src,
                  size_t len,
                  mbstate_t * ps);
```

Arguments:

dst A pointer to a buffer where the function can store the multibyte-character string.
src A pointer to the wide-character string that you want to convert.
len The maximum number of multibyte characters to store.
ps An internal pointer that lets wcsrtombs() be a restartable version of wcstombs(); if ps is NULL, wcsrtombs() uses its own internal variable.

You can call mbsinit() to determine the status of this variable.

Library:

```
lbc
```

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The wcsrtombs() function converts a string of wide-characters pointed to by src into the corresponding multi-byte characters pointed to by dst. No more than len bytes are stored, including the terminating NULL character.

The function converts each character as if by a call to wctomb() and stops early if:

- A sequence of bytes doesn’t conform to a valid character.
wcsrtombs()

- Converting the next character would exceed the limit of \( len \) total bytes.
  The wcsrtombs() function uses \( ps \) to make it thread safe; if \( ps \) is a NULL pointer, wcsrtombs() uses its own internal pointer.

Returns:

The number of total bytes successfully converted, not including the terminating NULL byte, or \((\text{size}_t) \cdot -1\) if an invalid wide-character code was found.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Count the wide characters at the beginning of a string that are in a given character set

Synopsis:

```
#include <wchar.h>

size_t wcsstrn( const wchar_t * ws1, const wchar_t * ws2 );
```

Arguments:

- `ws1`: The wide-character string that you want to search.
- `ws2`: The set of wide characters you want to look for.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsstrn()` function returns the length of the initial segment of the string pointed to by `ws1` consisting entirely of wide characters from the string pointed to by `ws2`. The terminating NUL isn’t considered to be part of `ws2`.

Returns:

The length of the segment.

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: Yes

continued...
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strstr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcsstr()`
wcsstr()  
Find one wide-character string inside another

Synopsis:

```c
#include <wchar.h>

wchar_t * wcsstr( const wchar_t * ws1,  
                  const wchar_t * ws2 );
```

Arguments:

- `ws1` The wide-character string that you want to search.
- `ws2` The wide-character string that you’re looking for.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcsstr()` function locates the first occurrence in the string pointed to by `ws1` of the sequence of wide characters, excluding the terminating NUL, in the string pointed to by `ws2`.

Returns:

A pointer to the located string, NULL if the string wasn’t found, or the same pointer as `ws1` if `ws2` points to a zero-length string.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
wcsstr()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memchr(), strchr(), strcspn(), strpbrk(), strrchr(), strspn(), strstr(), strtok(), strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcsrchr(), wcsspn(), wcstok()
Synopsis:

```
#include <wchar.h>

double wcstod( const wchar_t * ptr,
               wchar_t ** endptr );

float wcstof( const wchar_t * ptr,
              wchar ** endptr );

long double wcstold( const wchar_t * ptr,
                     wchar ** endptr );
```

Arguments:

- `nptr` A pointer to the string to parse.
- `endptr` If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

These functions convert a wide-character string to a number:

- `wcstod()` function converts it to a `double`
- `wcstof()` converts it to a `float`
- `wcstold()` to a `long double`.

These functions recognize strings containing the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits containing an optional decimal point
- an optional e or E, followed by an optionally signed sequence of digits.

The functions expect the string to have a plus or minus sign, followed by one of these forms:

- A sequence of decimal digits, optionally followed by a radix character, optionally followed by an exponent part.
- A 0x or 0X followed by a sequence of hexadecimal digits, optionally followed by a radix character, optionally followed by a binary exponent part.
- The case-insensitive string INF or INFINITY.
- The case-insensitive string NAN or NAN (n-wchar-sequence) where n-wchar-sequence may be a digit, a nondigit, a n-wchar-sequence digit or a n-wchar-sequence nondigit.

The value is correctly rounded if the subject is hexadecimal and FLT_RADIX is 2.

The radix character is locale specific, depending upon LC_NUMERIC.

The conversion ends at the first unrecognized character. If endptr isn’t NULL, a pointer to the unrecognized wide character is stored in the object endptr points to.

Because 0 is a valid return that is also used for an error, you should set errno to 0 before calling these functions, and check errno again afterward. These functions don’t change errno on success.

**Returns:**

The converted value. If the correct value would cause overflow, plus or minus HUGE_VAL is returned according to the sign, and errno is set to ERANGE. If the correct value would cause underflow, then zero is returned, and errno is set to ERANGE.
Zero is returned when the input string can’t be converted. When an error occurs, *errno* indicates the error detected.

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

*errno*

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**wcstolmax(), wcstoulmax()**  © 2007, QNX Software Systems GmbH & Co. KG.

Convert a wide-character string into an integer

**Synopsis:**

```c
#include <inttypes.h>

intmax_t wcstolmax ( const wchar_t * nptr,
                    wchar_t ** endptr,
                    int base );

uintmax_t wcstoulmax ( const wchar_t * nptr,
                      wchar_t ** endptr,
                      int base );
```

**Arguments:**

- `nptr`  A pointer to the string to parse.
- `endptr` If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.
- `base`  The base of the number being parsed:
  - If `base` is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are `0x` or `0X` the digits are treated as hexadecimal. If the first character is `0`, the digits are treated as octal. Otherwise, the digits are treated as decimal.
  - If `base` isn’t zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than `base` are permitted. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits.

**Library:**

libc

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The `wcstoimax()` and `wcstoumax()` functions are the same as the `wcstol()`, `wcstoll()`, `wcstoul()`, and `wcstoull()` functions except that they return objects of type `intmax_t` and `uintmax_t`.

Returns:

The converted value.

If the correct value causes an overflow, (INTMAX_MAX | UINTMAX_MAX or INTMAX_MIN) is returned according to the sign and `errno` is set to ERANGE. If `base` is out of range, zero is returned and `errno` is set to EINVAL.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`strtol()`, `wcstol()`, `wcstoul()`

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**Synopsis:**

```c
#include <wchar.h>

wchar_t * wcstok( wchar_t * ws1, const wchar_t * ws2, wchar_t ** ptr );
```

**Arguments:**

- *ws1*  NULL, or the wide-character string that you want to break into tokens; see below.
- *ws2*  A set of the wide characters that separate the tokens.
- *ptr*  The address of a pointer to a wchar_t object, which the function can use to store information necessary for it to continue scanning the same string.

**Library:**

```none
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The function `wcstok()` breaks the wide-character string pointed to by *ws1* into a sequence of tokens, each of which is delimited by a wide character from the string pointed to by *ws2*.

In the first call to `wcstok()`, *ws1* must point to a null-terminated string, *ws2* must point to a null-terminated string of separator wide characters, and *ptr* is ignored. The `wcstok()` function returns a pointer to the first wide character of the first token, writes a NUL wide character into *ws1* immediately following the returned token, and updates *ptr*.

In subsequent calls, *ws1* must be NULL, and *ptr* must be unchanged from the previous call so that subsequent calls will move through the string *ws1*, returning successive tokens until no tokens remain. The
separator string \texttt{ws2} may differ from call to call. When no tokens remain in \texttt{ws1}, a NULL pointer is returned.

**Returns:**

A pointer to the token found, or NULL if no token was found.

**Classification:**

ANSI, POSIX 1003.1

**\texttt{wcstok()}**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

\texttt{memchr()}, \texttt{strchr()}, \texttt{strcspn()}, \texttt{strpbrk()}, \texttt{strrchr()}, \texttt{strset()}, \texttt{strspn()}, \texttt{strstr()}, \texttt{strtok()}, \texttt{strtok\_r()}, \texttt{wcschr()}, \texttt{wcscespn()}, \texttt{wcspbrk()}, \texttt{wcsrchr()}, \texttt{wcsspn()}, \texttt{wcsstr()}
**wcstol(), wcstoll()**

Convert a wide-character string into a long integer

**Synopsis:**
```
#include <stdlib.h>

long wcstol( const wchar_t * ptr,
             wchar_t ** endptr,
             int base );

long long wcstoll( const wchar_t * ptr,
                   wchar_t ** endptr,
                   int base );
```

**Arguments:**
- **ptr** A pointer to the string to parse.
- **endptr** If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.
- **base** The base of the number being parsed:
  - If `base` is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
  - If `base` isn’t zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

**Library:**
```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

The \texttt{wcstol()} function converts the string pointed to by \textit{ptr} into a \texttt{long}; \texttt{wcstoll()} converts the string into a \texttt{long long}.

These functions recognize strings that contain the following:

- optional white space
- an optional plus or minus sign
- a sequence of digits and letters.

The conversion ends at the first unrecognized wide character. If \textit{endptr} isn’t NULL, a pointer to the unrecognized wide character is stored in the object \textit{endptr} points to.

Returns:

The converted value.

If the correct value causes an overflow, \texttt{LONG\_MAX | LONGLONG\_MAX} or \texttt{LONG\_MIN | LONGLONG\_MIN} is returned according to the sign, and \texttt{errno} is set to ERANGE. If \textit{base} is out of range, zero is returned and \texttt{errno} is set to EDOM.

Errors:

- \texttt{ERANGE} The value is not representable
- \texttt{EINVAL} The value for \textit{base} is not supported or no conversion could be performed.

Classification:

- ANSI, POSIX 1003.1

Safety

Cancellation point: No

continued…
**wcstol(), wcstoll()**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*errno*

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
wcstombs()

Convert a wide-character string into a multibyte character string

Synopsis:

```c
#include <stdlib.h>

size_t wcstombs( char* s,
                 const wchar_t* pwcs,
                 size_t n );
```

Arguments:

- `s` A pointer to a buffer where the function can store the multibyte-character string.
- `pwcs` The wide-character string that you want to convert.
- `n` The maximum number of bytes to store.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcstombs()` function converts a sequence of wide character codes from the array pointed to by `pwcs` into a sequence of multibyte characters, and stores them in the array pointed to by `s`. It stops if a multibyte character exceeds the limit of `n` total bytes, or if the NUL character is stored. At most `n` bytes of the array pointed to by `s` are modified.

The `wcsrtombs()` function is a restartable version of `wcstombs()`.

Returns:

The number of array elements modified, not including the terminating zero code, if present, or `(size_t)-1` if an invalid multibyte character is encountered.
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

wchar_t wbuffer[] = {
  0x0073,
  0x0074,
  0x0072,
  0x0069,
  0x006e,
  0x0067,
  0x0000
};

int main( void )
{
  char mbsbuffer[50];
  int i, len;

  len = wcstombs( mbsbuffer, wbuffer, 50 );
  if( len != -1 ) {
    for( i = 0; i < len; i++ )
      printf( "/%4.4x", wbuffer[i] );
    printf( "\n" );
    mbsbuffer[len] = '\0';
    printf( "%s(%d)\n", mbsbuffer, len );
  }
  return EXIT_SUCCESS;
}
```

produces the output:

```
/0073/0074/0072/0069/006e/0067
string(6)
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: No

...continued
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

mblen(), mbtowc(), mbstowcs(), wcsrtombs(), wctomb()
Convert a wide-character string into an unsigned long integer

**Synopsis:**

```c
#include <stdlib.h>

long wcstoul( const wchar_t * ptr,
              wchar_t ** endptr,
              int base );

long long wcstoull( const wchar_t * ptr,
                    char** endptr,
                    int base );
```

**Arguments:**

- **ptr** A pointer to the string to parse.
- **endptr** If this argument isn’t NULL, the function stores in it a pointer to the first unrecognized character found in the string.
- **base** The base of the number being parsed:
  - If `base` is zero, the first characters after the optional sign determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.
  - If `base` isn’t zero, it must have a value between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

**Library:**

- **libc**

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.
Description:

These functions convert a wide-character string into a number:

- \texttt{wcstoul()} converts the string into an \texttt{unsigned long}
- \texttt{wcstoull()} converts it into a \texttt{unsigned long long}.

These functions recognize a string containing optional white space, followed by a sequence of digits and letters. The conversion ends at the first unrecognized character. A pointer to that character is stored in the object \texttt{endptr} points to, if \texttt{endptr} isn’t NULL.

If \texttt{base} is zero, the first characters determine the base used for the conversion. If the first characters are 0x or 0X the digits are treated as hexadecimal. If the first character is 0, the digits are treated as octal. Otherwise, the digits are treated as decimal.

If \texttt{base} isn’t zero, it must have a value of between 2 and 36. The letters a-z and A-Z represent the values 10 through 35. Only those letters whose designated values are less than \texttt{base} are permitted. If the value of \texttt{base} is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits.

Returns:

The converted value.

If the correct value causes an overflow, \texttt{ULONG_MAX} | \texttt{ULONGLONG_MAX} is returned and \texttt{errno} is set to \texttt{ERANGE}. If \texttt{base} is out of range, zero is returned and \texttt{errno} is set to \texttt{EDOM}.

Errors:

- \texttt{ERANGE} The value is not representable
- \texttt{EINV AL} The value for \texttt{base} is not supported or no conversion could be performed.
**Classification:**

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

errno

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Transform one wide-character string into another, to a given length

Synopsis:

```c
#include <wchar.h>

int wcscfrm( wchar_t *ws1,
             const wchar_t *ws2,
             size_t n );
```

Arguments:

- `ws1` The string that you want to transform.
- `ws2` The string that you want to place in `dst`.
- `n` The maximum number of characters to transform.

Library:

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wcscfrm()` function transforms the string pointed to by `ws2` to the buffer pointed to by `ws1`, to a maximum of `n` wide-characters, including the terminating null. The two strings shouldn’t overlap.

A call to `wcscmp()` returns the same result for two strings transformed by `wcscfrm()` as `wcscoll()` would return for the original versions of the strings.

This function doesn’t report errors in its returns; set `errno` to 0, call `wcscfrm()`, and then check `errno` again.

Returns:

The length of the transformed wide-character string. If this value is greater than `n`, the contents of `ws1` are indeterminate.
**wcscxfrm()**

**Classification:**

ANSI, POSIX 1003.1

**wcscxfrm()**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

*setlocale(), strxfrm()*
Synopsis:

```c
#include <wchar.h>

int wctob( wint_t c );
```

Arguments:

- `c`  The wide character that you want to convert.

Library:

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wctob()` function returns the single-byte representation of a wide character.

This function is affected by LC_CTYPE.

Returns:

The single-byte representation, or EOF if `c` isn’t a valid single-byte character.

Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
wctomb()

Convert a wide character into a multibyte character

Synopsis:

```c
#include <stdlib.h>
int wctomb( char * s,
            wchar_t wc );
```

Arguments:

- `s`  NULL, or a pointer to a location where the function can store the multibyte character.
- `wc`  The wide character that you want to convert.

Library:

`libc`

Use the -l c option to qcc to link against this library. This library is usually included automatically.

Description:

The `wctomb()` function determines the number of bytes required to represent the multibyte character corresponding to the code contained in `wc`. If `s` isn’t NULL, the multibyte character representation is stored in the array it points to. At most MB_CUR_MAX characters are stored.

Returns:

- If `s` is NULL:
  
  0  The `wctomb()` function uses locale specific multibyte character encoding that’s not state-dependent.
  
  >0  The function is state-dependent.

- If `s` isn’t NULL:
  
  -1  If the value of `wchar` doesn’t correspond to a valid multibyte character.
  
  `x`  The number of bytes that comprise the multibyte character corresponding to the value of `wchar`. 
Examples:

```c
#include <stdio.h>
#include <stdlib.h>

wchar_t wchar = { 0x0073 };  
char mbuffer[MB_CUR_MAX];  

int main( void )
{
    int len;

    printf( "Character encodings do %shave "  
            "state-dependent \nencoding,\n", 
            ( wctomb( NULL, 0 ) )  
            ? "" : "not " );

    len = wctomb( mbuffer, wchar );
    mbuffer[len] = '\0';  
    printf( "%s(%d)\n", mbuffer, len );
    return EXIT_SUCCESS;
}
```

This produces the output:

Character encodings do not have state-dependent encoding.

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**wctrans()**

*Define a wide-character mapping*

**Synopsis:**

```c
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

**Arguments:**

- `property` The type of mapping; see below.

**Library:**

- `libc`

  Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wctrans()` function determines a mapping rule for wide-character codes according to the category `LC_CTYPE`, particularly for use with `towctrans()`.

The following mappings are defined in all locales, although additional classes may be defined for `LC_CTYPE`:

- `tolower`
- `toupper`

Use `setlocale()` to modify the category `LC_CTYPE`.

**Returns:**

An object that you can use in a call to `towctrans()`, or 0 if the specified character mapping isn’t valid for the current locale.

**Classification:**

ANSI, POSIX 1003.1
### Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

- `towctrans()`
- String manipulation functions
- Wide-character functions
**wctype()**

*Define a wide-character class*

**Synopsis:**

```c
#include <wctype.h>

wctype_t wctype( const char * property );
```

**Arguments:**

`property`  
A string that defines the property of the class; see below.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wctype()` function determines a classification rule for wide-character codes according to the category LC_CTYPE, particularly for use with `iswctype()`.

Some classes are defined in all locales, although additional classes may be defined for LC_CTYPE. Use `setlocale()` to modify the category LC_CTYPE.

Defined Classes:

- `alnum`
- `alpha`
- `blank`
- `cntrl`
- `digit`
- `graph`
- `lower`
- `print`
- `punct`
- `space`
- `upper`
- `xdigit`

**Returns:**

A `wctype_t` object that you can use in a call to `iswctype()`, or 0 if the character class name isn’t valid for the current locale.
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

setlocale()

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
wmemchr() © 2007, QNX Software Systems GmbH & Co. KG.

Locate the first occurrence of a wide character in a buffer

Synopsis:

```c
#include <wchar.h>

wchar_t * wmemchr( const wchar_t * ws,
                     wchar_t wc,
                     size_t n );
```

Arguments:

- `ws` The buffer that you want to search.
- `wc` The character that you’re looking for.
- `n` The number of wide characters to search in the buffer.

Library:

`libc`

Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `wmemchr()` function locates the first occurrence of `wc` in the first `n` wide characters of the buffer pointed to by `ws`.

The `wmemchr()` function is locale-independent and treats all `wchar_t` values identically, even if they’re null or invalid characters.

Returns:

A pointer to the located character, or NULL if `wc` couldn’t be found.

Classification:

ANSI, POSIX 1003.1
wmemchr()

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`memcpy()`, `memcmp()`, `memcsnpy()`, `memicmp()`, `memmove()`, `memset()`

`wcschr()`, `wcsrchr()`, `wmemccpy()`, `wmemcmp()`, `wmemcpy()`, `wmemmove()`,

`wmemset()`


**wmemcmp()**

*Compare the wide characters in two buffers*

**Synopsis:**

```c
#include <wchar.h>

int wmemcmp( const wchar_t * ws1,  
             const wchar_t * ws2,  
             size_t n );
```

**Arguments:**

- `ws1`, `ws2` The wide-character strings that you want to compare.
- `n` The number of wide characters to compare.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wmemcmp()` function compares `n` wide characters of the buffer pointed to by `ws1` to those in the buffer pointed to by `ws2`.

**Returns:**

- `<0` `ws1` is less than `ws2`.
- `0` `ws1` is equal to `ws2`.
- `>0` `ws1` is greater than `ws2`.

**Classification:**

ANSI, POSIX 1003.1
wmemcmp()  

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memccpy(), memcmp(), memcpy(), memicmp(), memmove(), memset()
wcsncmp(), wcscmp(), wmemchr(), wmemcmp(), wmemmove(), wmemset()
Copy wide characters from one buffer to another

Synopsis:
```
#include <wchar.h>

wchar_t * wmemcpy( wchar_t * ws1, const wchar_t * ws2, size_t n );
```

Arguments:
- `ws1`: A pointer to the buffer that you want to copy the wide characters into.
- `ws2`: A pointer to the buffer that you want to copy the wide characters from.
- `n`: The number of wide characters to copy.

Library:
```
libc
```
Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

Description:
The `wmemcpy()` function copies `n` wide characters from the buffer pointed to by `ws2` into the buffer pointed to by `ws1`.

The `wmemcpy()` function is locale-independent and treats all `wchar_t` values identically, even if they’re null or invalid characters.

Copying overlapping buffers isn’t guaranteed to work; use `wmemmove()` to copy buffers that overlap.

Returns:
A pointer to the destination buffer (i.e. the same pointer as `ws1`).
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memccpy(), memcmp(), memcpy(), memicmp(), memmove(), memset()
wcsncpy(), wmemchr(), wmemcmp(), wmemmove(), wmemset()
**wmemmove()**

*Copy wide characters from one buffer to another*

**Synopsis:**

```c
#include <wchar.h>

wchar_t * wmemmove( wchar_t * ws1,
                     const wchar_t * ws2,
                     size_t n );
```

**Arguments:**

- `ws1` A pointer to where you want the function to copy the data.
- `ws2` A pointer to the buffer that you want to copy data from.
- `n` The number of wide characters to copy.

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `memmove()` function copies `n` wide characters from the buffer pointed to by `ws2` to the buffer pointed to by `ws1`. This function copies overlapping regions safely.

The `wmemmove()` function is locale-independent and treats all `wchar_t` values identically, even if they’re null or invalid characters.

Use `wmemcpy()` for greater speed when copying buffers that don’t overlap.

**Returns:**

A pointer to the destination buffer (i.e. the same pointed as `ws1`).

---

© 2007, QNX Software Systems GmbH & Co. KG.
Classification:

ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memccpy(), memcmp(), memcpy(), memicmp(), memmove(), memset()
wmemchr(), wmemcmp(), wmemcpy(), wmemset()
Set wide characters in memory

Synopsis:

```c
#include <wchar.h>

wchar_t * wmemset( wchar_t * ws,
                   wchar_t wc,
                   size_t n)
```

Arguments:

- `ws`: A pointer to the memory that you want to set.
- `wc`: The value that you want to store in each wide character.
- `length`: The number of wide characters to set.

Library:

```
libc
```

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `memset()` function fills `n` wide characters starting at `ws` with the value `wc`.

The `wmemset()` function is locale-independent and treats all `wchar_t` values identically, even if they’re null or invalid wide characters.

Returns:

A pointer to the destination buffer (i.e. the same pointer as `ws`).

Classification:

ANSI, POSIX 1003.1
Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

memccpy(), memcmp(), memcpy(), memicmp(), memmove(), memset(), wmemchr(), wmemcmp(), wmemcpy(), wmemmove()
wordexp()
Perform word expansions

Synopsis:
#include <wordexp.h>

int wordexp( const char * words,
              wordexp_t * pwordexp,
              int flags );

Library:
libc
Use the -l c option to qcc to link against this library. This library is usually included automatically.

This function is in libc.a, but not in libc.so (in order to save space).

Description:
The C bindings for performing word expansions aren’t currently supported.

Returns:
-1 to indicate an error (errno is set).

Errors:
WRDE_NOSYS
The wordexp() function isn’t currently supported.

Classification:
POSIX 1003.1
wordexp()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

glob(), globfree(), wordfree()
**wordfree()**

Free a word expansion buffer

**Synopsis:**

```c
#include <wordexp.h>

void wordfree( wordexp_t * pwordexp );
```

**Library:**

`libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The C bindings for performing word expansions aren't currently supported.

**Classification:**

POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`glob()`, `globfree()`, `wordexp()`
**Synopsis:**

```c
#include <wchar.h>

int wprintf( const char* format,
             ... );
```

**Arguments:**

- **format**  
  A wide-character string that specifies the format of the output. The formatting string determines what additional arguments you need to provide. For more information, see `printf()`.

**Library:**

- **libc**

  Use the `-lc` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `wprintf()` function writes output to the `stdout` stream, under control of the argument `format`. It’s the wide-character version of `printf()`.

**Returns:**

The number of characters written, or a negative value if an output error occurred (`errno` is set).

**Classification:**

- ANSI, POSIX 1003.1

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Yes</td>
</tr>
</tbody>
</table>

continued…
wprintf()

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

errno, fprintf(), fwprintf(), printf(), sprintf(), swprintf(), vfprintf(), vfprintf(), vfprintf(), vsprintf(), vsprintf(), vsprintf(), vswprintf(), vwprintf()
write()
Write bytes to a file

Synopsis:
#include <unistd.h>

ssize_t write( int fildes,
               const void* buf,
               size_t nbyte );

Arguments:
  fildes    The file descriptor for the file you want to write in.
  buf       A pointer to a buffer that contains the data you want to
             write.
  nbyte     The number of bytes to write.

Library:
  libc
Use the -l c option to qcc to link against this library. This library is
usually included automatically.

Description:
The write() function attempts to write nbyte bytes to the file associated
with the open file descriptor, fildes, from the buffer pointed to by buf.
If nbyte is zero, write() returns zero, and has no other effect.

On a regular file or other file capable of seeking, and if O_APPEND
isn’t set, write() starts at a position in the file given by the file offset
associated with fildes. If O_APPEND is set, the file offset is set to the
end of file before each write operation. Before successfully returning
from write(), the file offset is incremented by the number of bytes
actually written. On a regular file, if this incremented file offset is
greater than the length of the file, the length of the file is set to this file
offset.
Note that the `write()` call ignores advisory locks that may have been set by the `fcntl()` function.

On a file not capable of seeking, `write()` starts at the current position.

If `write()` requests that more bytes be written than there’s room for (for example, all blocks on a disk are already allocated), only as many bytes as there’s room for are written. For example, if there’s only room for 80 more bytes in a file, a write of 512 bytes would return 80. The next write of a nonzero number of bytes would give a failure return (except as noted below).

When `write()` returns successfully, its return value is the number of bytes actually written to the file. This number is never greater then `nbyte`, although it may be less than `nbyte` under certain circumstances detailed below.

If `write()` is interrupted by a signal before it has written any data, it returns a value of -1, and `errno` is set to EINTR. However, if `write()` is interrupted by a signal after it has successfully written some data, it returns the number of bytes written.

If the value of `nbyte` is greater than INT_MAX, `write()` returns -1 and sets `errno` to EINVAL. See `<limits.h>`.

Write requests to a pipe (or FIFO) are handled the same as a regular file, with the following exceptions:

- There’s no file offset associated with a pipe, therefore each write request appends to the end of the pipe.

- Write requests of PIPE_BUF bytes or less aren’t interleaved with data from other processes doing writes on the same pipe. Writes of greater than PIPE_BUF bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag is set.

- If the O_NONBLOCK flag is clear, a write request may cause the process to block, but on normal completion it returns `nbyte`. 
write()

- If the O_NONBLOCK flag is set, write requests are handled differently, in the following ways:
  - The write() function doesn’t block the process.
  - Write requests for PIPE_BUF bytes or less either succeed completely and return nbyte, or return -1 and errno is set to EAGAIN.

If you call write() with nbyte greater than PIPE_BUF bytes, it either transfers what it can and returns the number of bytes written, or transfers no data, returning -1 and setting errno to EAGAIN. Also, if nbyte is greater than PIPE_BUF bytes and all data previously written to the pipe has been read (that is, the pipe is empty), write() transfers at least PIPE_BUF bytes.

When attempting to write to a file (other than a pipe or FIFO) that supports nonblocking writes and can’t accept the data immediately:

- If the O_NONBLOCK flag is clear, write() blocks until the data can be accepted.
- If the O_NONBLOCK flag is set, write() doesn’t block the process. If some data can be written without blocking the process, write() transfers what it can and returns the number of bytes written. Otherwise, it returns -1 and sets errno to EAGAIN.

If write() is called with the file offset beyond the end-of-file, the file is extended to the current file offset with the intervening bytes filled with zeroes. This is a useful technique for pregrowing a file.

If write() succeeds, the st_csize and st_mtime fields of the file are marked for update.

Returns:

The number of bytes written, or -1 if an error occurred (errno is set).
Errors:

EAGAIN  The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the write operation.

EBADF   The file descriptor, fildes, isn’t a valid file descriptor open for writing.

EFBIG   The file is a regular file, where nbytes is greater than 0, and the starting position is greater than or equal to the offset maximum associated with the file.

EINTR   The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.

EIO     A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.

ENOSPC  There’s no free space remaining on the device containing the file.

ENOSYS  The write() function isn’t implemented for the filesystem specified by fildes.

EPIPE   An attempt was made to write to a pipe (or FIFO) that isn’t open for reading by any process. A SIGPIPE signal is also sent to the process.

Examples:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

char buffer[] = { "A text record to be written" };

int main( void )
{
    int fd;
    int size_written;
```
/* open a file for output */
/* replace existing file if it exists */
fd = creat( "myfile.dat", S_IRUSR | S_IWUSR );

/* write the text */
size_written = write( fd, buffer, sizeof( buffer ) );

/* test for error */
if( size_written != sizeof( buffer ) ) {
    perror( "Error writing myfile.dat" );
    return EXIT_FAILURE;
}

/* close the file */
close( fd );

return EXIT_SUCCESS;

Classification:
POSIX 1003.1 XSI

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

close(), creat(), dup(), dup2(), errno, fcntl(), lseek(), open(), pipe(), read(), readv(), select(), writev()
**writeblock()**

Write blocks of data to a file

**Synopsis:**

```c
#include <unistd.h>

int writeblock( int fd,  
    size_t blksize, 
    unsigned block, 
    int numblks, 
    const void *buff );
```

**Arguments:**

- `fd`  The file descriptor for the file you want to write in.
- `blksize`  The number of bytes in each block of data.
- `block`  The block number from which to start writing. Blocks are numbered starting at 0.
- `numblks`  The number of blocks to write.
- `buff`  A pointer to a buffer that contains the blocks of data that you want to write.

**Library:**

- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**

The `writeblock()` function writes `numblks` blocks of data to the file associated with the open file descriptor, `fd`, from the buffer pointed to by `buff`, starting at block number `block`.

This function is useful for direct updating of raw blocks on a block special device (for example, raw disk blocks), but you can also use it for high-speed updating (for example, of database files). The speed gain is through the combined seek/write implicit in this call.

If `numblks` is zero, `writeblock()` returns zero, and has no other results.
writeblock()

If successful, writeblock() returns the number of blocks actually written to the disk associated with fd. This number is never greater than numblks, but could be less than numblks if one of the following occurs:

- The process attempts to write more blocks than implementation limits allow to be written in a single atomic operation.

- A write error occurred after writing at least one block, and you set one of the sync flags (O_SYNC or O_DSYNC — see open()) when you opened the file.

If a write error occurs on the first block and one of the sync flags is set, writeblock() returns -1 and sets errno to EIO.

If one of the sync flags is set, writeblock() doesn’t return until the blocks are actually transferred to the disk. If neither of the flags is set, writeblock() places the blocks in the cache and schedules them for writing as soon as possible, but returns before the writing takes place.

In the latter instance, it’s impossible for the application to know if the write succeeded or not (due to system failures or bad disk blocks). Using the sync flags significantly impacts the performance of writeblock(), but guarantees that the data can be recovered.

Returns:

The number of blocks actually written. If an error occurred, writeblock() returns -1, sets errno to indicate the error, and doesn’t change the contents of the buffer pointed to by buff.

Errors:

EBADF The fd argument isn’t a valid file descriptor that’s open for writing a block-oriented device.

EIO A physical write error occurred on the first block, and either O_DSYNC or O_SYNC is set.
writeblock()

EINVAL  The starting position is invalid (0 or negative), or beyond the end of the file.

Classification:

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

open(), readblock(), write()
writev()
Write bytes to a file

Synopsis:

```
#include <sys/uio.h>

ssize_t writev( int fildes,
                const iov_t* iov,
                int iovcnt );
```

Arguments:

- `fildes` The file descriptor for the file you want to write in.
- `iov` An array of `iov_t` objects that contain the data that you want to write.
- `iovcnt` The number of elements in the array.

Library:

```
libc
```
Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `writev()` function performs the same action as `write()`, but gathers the output data from the `iovcnt` buffers specified by the members of the `iov` array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt-1]`.

For `writev()`, the `iov_t` structure contains the following members:

- `iov_base` Base address of a memory area from which data should be written.
- `iov_len` The length of the memory area.

The `writev()` function always writes a complete area before proceeding to the next.

The maximum number of entries in the `iov` array is `UIO_MAXIOV`.
Note that `writev()` ignores advisory locks that may have been set by the `fcntl()` function.

If `writev()` is interrupted by a signal before it has written any data, it returns a value of -1, and `errno` is set to EINTR. However, if `writev()` is interrupted by a signal after it has successfully written some data, it will return the number of bytes written.

For more details, see the `write()` function.

**Returns:**

The number of bytes written, or -1 if an error occurs (`errno` is set).

**Errors:**

- **EAGAIN** The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the write operation.
- **EBADF** The file descriptor, `fildes`, isn’t a valid file descriptor open for writing.
- **EFBIG** The file is a regular file, where ` nbytes` is greater than 0, and the starting position is greater than or equal to the offset maximum associated with the file.
- **EINTR** The write operation was interrupted by a signal, and either no data was transferred, or the resource manager responsible for that file doesn’t report partial transfers.
- **EINVAL** The `iovcnt` argument is less than or equal to 0, or greater than UIO_MAXIOV.
- **EIO** A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.
- **ENOSPC** There is no free space remaining on the device containing the file.
- **ENOSYS** The `write()` function isn’t implemented for the filesystem specified by `filedes`. 
writev()

EPIPE  An attempt was made to write to a pipe (or FIFO) that isn’t open for reading by any process. A SIGPIPE signal is also sent to the process.

Classification:

POSIX 1003.1 XSI

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

close(), creat(), dup(), dup2(), errno, fcntl(), lseek(), open(), pipe(), read(), readv(), select(), write()
**wscanf()**  
Scan formatted wide-character input from stdin

**Synopsis:**
```
#include <wchar.h>

int wscanf( const char * format, ...
   );
```

**Arguments:**
- `format` A wide-character string that specifies the format of the input. For more information, see `scanf()`. The formatting string determines what additional arguments you need to provide.

**Library:**
- `libc`

Use the `-l c` option to `qcc` to link against this library. This library is usually included automatically.

**Description:**
The `wscanf()` function scans input from stdin under control of the `format` argument, assigning values to the remaining arguments. It is the wide-character version of `scanf()` and uses the same conversions.

**Returns:**
The number of input arguments for which values were successfully scanned and stored, or EOF if the scanning reached the end of the input stream before storing any values.

**Classification:**
- ANSI, POSIX 1003.1
Safety

Cancellation point: Yes
Interrupt handler: No
Signal handler: No
Thread: Yes

See also:

fscanf(), fwscanf(), scanf(), sscanf(), swscanf(), vfscanf(), vfwscanf(),
vscanf(), vsscanf(), vswscanf(), vwscanf()
Compute a Bessel function of the second kind

Synopsis:

```c
#include <math.h>

double y0( double x );

float y0f( float x );
```

Arguments:

- `x` The number that you want to compute the Bessel function for.

Library:

- `libbessel`

Use the `-l bessel` option to `qcc` to link against this library.

Description:

Compute the Bessel function of the second kind for `x`.

Returns:

The result of the Bessel function of `x`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Classification:

- `y0()` is POSIX 1003.1 XSI; `y0f()` is Unix

Safety

- Cancellation point: No

continued...
Safety

<table>
<thead>
<tr>
<th>Interrupt handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`errno, j0(), j1(), jn(), y1(), yn()`


**y1(), y1f()**

Compute a Bessel function of the second kind

**Synopsis:**

```c
#include <math.h>

double y1( double x );

float y1f( float x );
```

**Arguments:**

- `x` The number that you want to compute the Bessel function for.

**Library:**

- `libbessel`

  Use the `-l bessel` option to `qcc` to link against this library.

**Description:**

Compute the Bessel function of the second kind for `x`.

**Returns:**

The result of the Bessel function of `x`.

---

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don't change `errno` if no errors occurred.

**Examples:**

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;

    x = j0( 2.4 );
    y = yl( 1.58 );
}
```
```c
z = jn( 3, 2.4 );
printf( "j0(2.4) = %f, y1(1.58) = %f\n", x, y );
printf( "jn(3,2.4) = %f\n", z );
return EXIT_SUCCESS;
```

### Classification:

`y1()` is POSIX 1003.1 XSI; `y1f()` is Unix

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

### See also:

`errno`, `j0()`, `j1()`, `jn()`, `y0()`, `yn()`
yn(), ynf()

Compute a Bessel function of the second kind

Synopsis:

```c
#include <math.h>

double yn( int n,  
      double x );

float ynf( int n,  
      float x );
```

Arguments:

- `n`, `x`: The numbers that you want to compute the Bessel function for.

Library:

- `libbessel`

Use the `-l bessel` option to `qcc` to link against this library.

Description:

Compute the Bessel function of the second kind for `n` and `x`.

Returns:

The result of the Bessel function of `n` and `x`.

If an error occurs, these functions return 0, but this is also a valid mathematical result. If you want to check for errors, set `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.

Classification:

- `yn()` is POSIX 1003.1 XSI; `ynf()` is Unix
 yn(), ynf()

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

erno, j0(), j1(), jn(), y0(), y1()
Appendix A

SOCKS — A Basic Firewall

In this appendix...

About SOCKS 3811
How to SOCKSify a client 3811
What SOCKS expects 3812
About SOCKS

SOCKS is a package consisting of a proxy server, client programs (rftp and rtelnet), and a library (libsocks) for adapting other applications into new client programs.

The original SOCKS was written by David Koblas (koblas@netcom.com). The SOCKS protocol has changed over time. The client library shipped as of printing corresponds to SOCKS v4.2. Since the server and the clients must use the same SOCKS protocol, this library doesn’t work with servers of previous releases; clients compiled with these libraries won’t work with older servers.

How to SOCKSify a client

If your client is using UDP to transfer data, you can’t use SOCKS. To see if your client uses UDP, search for the string “SOCK_DGRAM” in your source.

1. At or near the beginning of main(), you can add a call to SOCKSinit().

   You can omit this step; the only reason for calling SOCKSinit() directly is to associate a name with your SOCKS client (rather than the generic “SOCKSclient” default string).

2. Add the following options to your compile commands:

   `-Dconnect=Rconnect -Dgetsockname=Rgetsockname \`
   `-Dbind=Rbind -Daccept=Raccept -Dlisten=Rlisten \`
   `-Drcmd=Rrcmd -Dselect=Rselect`

   If you’re using a Makefile, add these options to the definition of macro CFLAGS.

   These options replace calls to certain functions with versions that use the SOCKS server.
What SOCKS expects

<table>
<thead>
<tr>
<th>Non-SOCKS function</th>
<th>SOCKS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept()</td>
<td>Raccept()</td>
</tr>
<tr>
<td>bind()</td>
<td>Rbind()</td>
</tr>
<tr>
<td>connect()</td>
<td>Rconnect()</td>
</tr>
<tr>
<td>getsockname()</td>
<td>Rgetsockname()</td>
</tr>
<tr>
<td>listen()</td>
<td>Rlisten()</td>
</tr>
<tr>
<td>rcmd()</td>
<td>Rrcmd()</td>
</tr>
<tr>
<td>select()</td>
<td>Rselect()</td>
</tr>
</tbody>
</table>

3 Link against the SOCKS library by adding `-l socks` to your link line.

If you’re using a `Makefile`, simply add this information to the definition of the macro `LDFLAGS`.

For most programs, the above steps should be sufficient to SOCKSify the package. If the above doesn’t work, you may need to look at things a little more closely. The next section describes how the SOCKS library expects to be used.

What SOCKS expects

The SOCKS library covers only some of the socket functions, which must be called in a particular order:

1 The first socket function invoked must be either `connect()` or `rcmd()`.

2 If you call `connect()` on a nonblocking socket, no I/O can occur on that socket until another `connect()`, with the same arguments, returns `-1` and sets `errno` to `EISCONN`. This is required even if you use `select()` on write to check the readiness of that socket.
While a connection is still pending, don’t try to start another connection via `connect()`, or start a sequence of `bind()`, `getsockname()`, `listen()`, and `accept()`.

3 You must call `bind()` after a successful `connect()` call to a host for a specific service.

4 You must follow the call to `bind()` by calls to `getsockname()`, `listen()`, and `accept()`, in that order.

Most client programs fit these assumptions very well and can be SOCKSified without changing the code at all using the steps described in “How to SOCKSify a client.”

Some client programs use a `bind()` before each `connect()`. If the `bind()` is used to claim a specific port or a specific network interface, the current SOCKS library can’t accommodate such use. Very often though, such a `bind()` call is there for no specific reason and may simply be deleted.
Appendix B

Summary of Safety Information

In this appendix...

Cancellation points 3817
Interrupt handlers 3822
Signal handlers 3825
Multithreaded programs 3839
Cancellation points

The following functions are cancellation points:

- `ConnectAttach()`
- `ConnectAttach_r()`
- `ConnectDetach()`
- `ConnectDetach_r()`
- `ConnectServerInfo()`
- `ConnectServerInfo_r()`
- `InterruptWait()`
- `InterruptWait_r()`
- `MsgSend()`
- `MsgSend_r()`
- `MsgSendsv()`
- `MsgSendsv_r()`
- `SignalSuspend()`
- `SignalSuspend_r()`
- `SyncCondvarSignal()`
- `SyncCondvarSignal_r()`
- `SyncCondvarWait()`
- `SyncCondvarWait_r()`
- `ThreadJoin()`
- `ThreadJoin_r()`
- `accept()`
- `aio_suspend()`
- `cache_finil`
- `cache_init()`
- `cfgopen()`
- `chsize()`
- `clock_nanosleep()`
- `close()`
- `closedir()`
- `closelog()`
- `connect()`
- `create()`
- `creator()`
- `dclose()`
- `dlclose()`
- `dlopen()`
- `ds_clear()`
- `ds_create()`
- `ds_deregister()`
- `ds_get()`
- `ds_register()`
- `ds_set()`
- `endgrent()`
- `endhostent()`
- `endnetent()`
- `endprotoent()`
- `endpwent()`
- `endservent()`
- `endspent()`
- `endutent()`
- `eof()`
- `err()`
- `errx()`
- `fcfgopen()`
- `fchown()`
- `fclose()`
Cancellation points

fcloseall()
fdopen()
fflush()
fgetc()
fgetchar()
fgets()
fgetspent()
fgetwc()
fgetws()
flushall()
fopen()
fopen64()
forkpty()
fprintf()
fputc()
fputchar()
fputs()
fputwc()
fputws()
freopen()
fscanf()
fsync()
ftell()
ftello()
ftw()
ftw64()
fwide()
fwprintf()
fwrite()
fwscanf()
getaddrinfo()
getchwd()
getgrent()
getgrgid()
getgrgid_r()
getgrnam()
getgrnam_r()
getgrouplist()
gethostbyaddr()
gethostbyaddr_r()
gethostbyname()
gethostbyname2()
gethostbyname_r()
gethostent()
gethostent_r()
gethostname()
gethostname_r()
getifaddrs()
getlogin()
getlogin_r()
getnameinfo()
gethostbyaddr()
gethostbyname()
gethostbyname_r()
gethostent()
gethostent_r()
gethostname()
gethostent_r()
getifaddrs()
getlogin()
getlogin_r()
getnameinfo()
gethostbyaddr()
gethostbyname()
gethostbyname_r()
gethostent()
gethostent_r()
gethostname()
gethostent_r()
getifaddrs()
getlogin()
getlogin_r()
getnameinfo()
gethostbyaddr()
gethostbyname()
gethostbyname_r()
gethostent()
gethostent_r()
gethostname()
getsockname()  mq_receive()
getsockopt()  mq_send()
getspent()  mq_timedreceive()
getspent_r()  mq_timedsend()
getspnam()  msync()
getspnam_r()  name_attach()
getutent()  name_close()
getutid()  name_detach()
getutline()  name_open()
getw()  nanosleep()
getwc()  nap()
getwchar()  napms()
getwd()  nbacoconnect()
glob()  nbacoconnect_result()
herror()  netmgr_ntostr()
if_indextoname()  netmgr_strtond()
if_nameindex()  nfs()
if_nameitomindex()  nfs64()
initstate()  open()
input_line()  open64()
iofunc_attr_lock()  opendir()
iofunc_attr_trylock()  openfd()
isfdtype()  openlog()
listen()  opempty()
listen_tty()  pathfind()
ltrunc()  pathfind_r()
message_attach()  pathmgr_symlink()
message_connect()  pathmgr_unlink()
message_detach()  pause()
mknod()  pccard_arm()
mkstemp()  pccard_attach()
mktemp()  pccard_detach()
modem_open()  pccard_info()
modem_read()  pccard_lock()
modem_script()  pccard_raw_read()
modem_write()  pccard_unlock()
mount()  pci_attach()
Cancellation points

pci_attach_device()            pulse_attach()
pci_detach()                  putc()
pci_detach_device()           putc_unlocked()
pci_find_class()              putchar()
pci_find_device()             putchar_unlocked()
pci_irq_routing_options()     puts()
pci_map_irq()                 puses()
ppci_present()                pututline()
p pci_read_config()            putw()
p pci_read_config16()          putwc()      
p pc_read_config32()           putwchar()    
p pci_read_config8()           pwrite()      
p pci_rescan_bus()             pwrite64()    
p pci_write_config()           rcmd()        
p pci_write_config16()         read()
pci_write_config32()           read_main_config_file() 
pci_write_config8()            readblock()    
pclose()                       readdir()     
perror()                       readv()       
poll()                         realpath()    
popent()                       recv()        
pread()                        recvfrom()    
pread64()                      recvmsg()     
printf()                       res_init()    
pthread_cond_timedwait()       res_mkquery()  
pthread_cond_wait()            res_query()    
pthread_join()                 res_querydomain() 
pthread_rwlock_rdlock()        res_search()   
pthread_rwlock_timedrdlock()   res_send()     
pthread_rwlock_timedwrlock()   resmgr_attach() 
pthread_rwlock_tryrdlock()     resmgr_block() 
pthread_rwlock_trywrlock()     resmgr_context_alloc() 
pthread_rwlock_wrlock()        resmgr_context_free() 
pthread_sleepon_lock()         resmgr_detach() 
pthread_sleepon_timedwait()    
pthread_sleepon_wait()         
pthread_testcancel()           

Cancellation points

resmgr_devino()  setsockopt()
resmgr_handler()  setutent()
rewind()  shm_ctl()
rewinddir()  shm_ctl_special()
resvport()  shutdown()
srscdbmgr_attach()  sigpause()
srscdbmgr_create()  sigsuspend()
srscdbmgr_destroy()  sigtimedwait()
srscdbmgr_detach()  sigwait()
srscdbmgr_devno_attach()  sigwaitinfo()
srscdbmgr_devno_detach()  sleep()
srscdbmgr_query()  slogb()
ruserok()  logf()
scandir()  logi()
scanf()  smack_close()
scctp_bindx()  smack_open()
scctp_connectx()  smack_read()
scctp_getladdr()  smack_send()
scctp_getpaddrs()  smack_timeout()
scctp_peeroff()  sockatmark()
scctp_recvmsg()  socket()
scctp_sendmsg()  socketpair()
seekdir()  sopen()
select_attach()  sopenfd()
select_detach()  sysctl()
sem_timedwait()  syslog()
sem_wait()  system()
send()  tcdrain()
sendmsg()  tell()
sendto() tell64()
setgrent()  telldir()
setgroups()  thread_pool_control()
sethostname()  thread_pool_destroy()
setnetent()  thread_pool_limits()
setservent()  thread_pool_start()
setpwnent()  tmpfile()
setservent()  tmpfile64()
Interrupt handlers

tmpnam()    vwarnx()
truncate()   vwscanf()
umount()    wait()
ungetc()     wait3()
ungetwc()    wait4()
unlink()     waitid()
usleep()     waitpid()
utmpname()   warn()
verr()       warnx()
verrnx()     wordexp()
vfscanf()     wprntf()
vfwscanf()    write()
vscanf()     write()     writeblock()
vslogf()     writev()    wscanff()
vwarn()     wscanff()    vwscanf()

See the “Caveats” section for the following functions for more information:

dispatch_handler()   spawnlpe()
fcntl()              spawnv()
spawnl()             spawnve()
spawnle()            spawnvp()
spawnlp()            spawnvpe()

Interrupt handlers

It’s safe to call the following functions from an interrupt handler:

CACHE_FLUSH()    ENDIAN_LE64()
CACHE_INVAL()    ENDIAN_RET16()
ENDIAN_BE16()    ENDIAN_RET32()
ENDIAN_BE32()    ENDIAN_RET64()
ENDIAN_BE64()    ENDIAN_SWAP16()
ENDIAN_LE16()    ENDIAN_SWAP32()
ENDIAN_LE32()    ENDIAN_SWAP64()
Interrupt handlers

GETIOVBASE()  basename()
GETIOVLEN()  bcmp()
InterruptDisable()  bcopy()
InterruptEnable()  bsearch()
InterruptLock()  bzero()
InterruptMask()  div()
InterruptUnlock()  gai_strerror()
InterruptUnmask()  htonl()
ND_NODE_CMP()  htons()
SETIOV()  hwi_find_item()
SYSPAGE_CPU_ENTRY()  hwi_find_tag()
SYSPAGE_ENTRY()  hwi_off2tag()
UNALIGNED_PUT16()  hwi_tag2off()
UNALIGNED_PUT32()  in16()
UNALIGNED_PUT64()  in16s()
UNALIGNED_RET16()  in32()
UNALIGNED_RET32()  in32s()
UNALIGNED_RET64()  in8()
_RESMGR_NPARTS()  in8s()
_RESMGR_PTR()  inbe16()
_RESMGR_STATUS()  inbe32()
abs()  inle16()
alphasort()  inle32()
atol()  ipsec_get_policylen()
atol()  ipsec_strerror()
atoll()  isalnum()
atol()  isalpha()
atomic_add()  isasciit()
atomic_add_value()  iscntrl()
atomic_clr()  isdigit()
atomic_clr_value()  isgraph()
atomic_set()  islower()
atomic_set_value()  isprint()
atomic_sub()  ispunct()
atomic_sub_value()  isspace()
atomic_toggle()  isupper()
atomic_toggle_value()  iswalnum()
Interrupt handlers

iswalpha() outbe32()
iswcntrl() outle16()
iswdigit() outle32()
iswgraph() rindex()
iswlower() setdomainname()
iswprint() sigaddset()
iswpunct() sigdelset()
iswspace() sigemptyset()
iswupper() sigfillset()
iswxdigit() sigismember()
isxdigit() sigmask()
itoa() straddstr()
lltoa() strcasecmp()
lsearch() strcat()
ltoa() strchr()
max() strcmp()
memccpy() strcmp()
memchr() strcoll()
memcmp() strcpy()
memcpy() strcspn()
memcpyv() stricmp()
memicmp() strlen()
memmove() strlwr()
memset() strcasecmp()
mini() strlen()
nanospin_count() strncmp()
nsec2timespec() strncpy()
ntohl() strnicmp()
ntohs() strnset()
offsetof() strpbrk()
out16() strrcchr()
out16s() strrev()
out32() strsep()
out32s() strset()
out8() strspn()
out8s() strstr()
outbe16() strtolmax()
Signal handlers

It’s safe to call the following functions from a signal handler:

- `strtok_r()`
- `strtol()`
- `strtoll()`
- `strtoul()`
- `strtoull()`
- `strtoumax()`
- `strupr()`
- `strxfrm()`
- `swab()`
- `timespec2nsec()`
- `tolower()`
- `toupper()`
- `towctrans()`
- `towlower()`
- `towupper()`
- `ulltoa()`
- `ultoa()`
- `utoa()`
- `va_arg()`
- `va_copy()`
- `va_end()`
- `va_start()`
- `wcscat()`
- `wcschr()`
- `wcsncmp()`
- `wcsncpy()`
- `wcspbrk()`
- `wcsrchr()`
- `wcscspn()`
- `wcslen()`
- `wcsncat()`
- `wcscmp()`
- `wcscpy()`
- `wcscsrn()`
- `wcstol()`
- `wcstoll()`
- `wcstoul()`
- `wcstoull()`
- `wcstoumax()`
- `wctrans()`
- `wctype()`
- `wmemchr()`
- `wmemcmp()`
- `wmemcpy()`
- `wmemmove()`
- `wmemset()`

See the “Caveats” section for the following functions for more information:

- `TraceEvent()`
- `nanospin_ns()`
- `nanospin_ns_to_count()`
CACHE_FLUSH()  ENDIAN_SWAP16()
CACHE_INVALID()  ENDIAN_SWAP32()
ChannelCreate()  ENDIAN_SWAP64()
ChannelCreate_r()  GETIOVBASE()
ChannelDestroy()  GETIOVLEN()
ChannelDestroy_r()  InterruptAttach()
ClockAdjust()  InterruptAttachEvent()
ClockAdjust_r()  InterruptAttachEvent_r()
ClockCycles()  InterruptAttach_r()
ClockId()  InterruptDetach()
ClockId_r()  InterruptDetach_r()
ClockPeriod()  InterruptDisable()
ClockPeriod_r()  InterruptEnable()
ClockTime()  InterruptHookIdle()
ClockTime_r()  InterruptHookTrace()
ConnectAttach()  InterruptLock()
ConnectAttach_r()  InterruptMask()
ConnectClientInfo()  InterruptUnlock()
ConnectClientInfo_r()  InterruptUnmask()
ConnectDetach()  InterruptWait()
ConnectDetach_r()  InterruptWait_r()
ConnectFlags()  MsgDeliverEvent()
ConnectFlags_r()  MsgDeliverEvent_r()
ConnectServerInfo()  MsgError()
ConnectServerInfo_r()  MsgError_r()
DebugBreak()  MsgInfo()
DebugKDBreak()  MsgInfo_r()
DebugKDOutput()  MsgKeyData()
ENDIAN_BE16()  MsgKeyData_r()
ENDIAN_BE32()  MsgRead()
ENDIAN_BE64()  MsgRead_r()
ENDIAN_LE16()  MsgReadv()
ENDIAN_LE32()  MsgReadv_r()
ENDIAN_LE64()  MsgReceive()
ENDIAN_RET16()  MsgReceivePulse()
ENDIAN_RET32()  MsgReceivePulse_r()
ENDIAN_RET64()  MsgReceivePulsev()
Signal handlers

MsgReceivePulse_r()
MsgReceive_r()
MsgReceivev()
MsgReceivev_r()
MsgReply()
MsgReply_r()
MsgReplyv()
MsgReplyv_r()
MsgReplyv_r()
MsgSend()
MsgSendPulse()
MsgSendPulse_r()
MsgSend_r()
MsgSendnc()
MsgSendnc_r()
MsgSendsv()
MsgSendsv_r()
MsgSendsv_r()
MsgSendsvnc()
MsgSendsvnc_r()
MsgSendv()
MsgSendv_r()
MsgSendvnc()
MsgSendvnc_r()
MsgSendvs()
MsgSendvs_r()
MsgSendvs_r()
MsgSendvsnc()
MsgSendvsnc_r()
MsgVerifyEvent()
MsgVerifyEvent_r()
MsgWrite()
MsgWrite_r()
MsgWritev()
MsgWritev_r()
ND_NODE_CMP()
SETIOV()
SYSPAGE_CPU_ENTRY()
SYSPAGE_ENTRY()
SchedGet()
Signal handlers

SyncTypeCreate() _intr_v86()
SyncTypeCreate_r() _sfree()
ThreadCancel() abs()
ThreadCancel_r() access()
ThreadCreate() aio_cancel()
ThreadCreate_r() aio_error()
ThreadCtl() aio_fsync()
ThreadCtl_r() aio_read()
ThreadDestroy() aio_return()
ThreadDestroy_r() aio_suspend()
ThreadDetach() aio_write()
ThreadDetach_r() alarm()
ThreadJoin() alloca()
ThreadJoin_r() alphasort()
TimerAlarm() asctime()
TimerAlarm_r() asctime_r()
TimerCreate() atof()
TimerCreate_r() atol()
TimerDestroy() atol()
TimerDestroy_r() atoll()
TimerInfo() atomic_add()
TimerInfo_r() atomic_add_value()
TimerSettime() atomic_clr()
TimerSettime_r() atomic_clr_value()
TimerTimeout() atomic_set()
TimerTimeout_r() atomic_set_value()
TraceEvent() atomic_sub()
UNALIGNED_PUT16() atomic_sub_value()
UNALIGNED_PUT32() atomic_toggle()
UNALIGNED_PUT64() atomic_toggle_value()
UNALIGNED_RET16() basename()
UNALIGNED_RET32() bcmp()
UNALIGNED_RET64() bcopy()
_RESMGR_NPARTS() bsearch()
_RESMGR_PTR() btowc()
_RESMGR_STATUS() bzerol()
_exit() cache_finil()
<table>
<thead>
<tr>
<th>Function</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_init()</td>
<td>ds_register()</td>
</tr>
<tr>
<td>cfgetispeed()</td>
<td>ds_set()</td>
</tr>
<tr>
<td>cfgetospeed()</td>
<td>dup()</td>
</tr>
<tr>
<td>cfgopen()</td>
<td>dup2()</td>
</tr>
<tr>
<td>cfmakeraw()</td>
<td>eaccess()</td>
</tr>
<tr>
<td>cfsetispeed()</td>
<td>encrypt()</td>
</tr>
<tr>
<td>cfsetospeed()</td>
<td>eof()</td>
</tr>
<tr>
<td>chdir()</td>
<td>err()</td>
</tr>
<tr>
<td>chmod()</td>
<td>errx()</td>
</tr>
<tr>
<td>chown()</td>
<td>execle()</td>
</tr>
<tr>
<td>chsize()</td>
<td>execve()</td>
</tr>
<tr>
<td>clock()</td>
<td>execvpe()</td>
</tr>
<tr>
<td>clock_getcpucontextid()</td>
<td>fcfgetopen()</td>
</tr>
<tr>
<td>clock_getres()</td>
<td>fchmod()</td>
</tr>
<tr>
<td>clock_gettime()</td>
<td>fchown()</td>
</tr>
<tr>
<td>clock_nanosleep()</td>
<td>fcntl()</td>
</tr>
<tr>
<td>clock_settime()</td>
<td>fdatasync()</td>
</tr>
<tr>
<td>close()</td>
<td>ffs()</td>
</tr>
<tr>
<td>confstr()</td>
<td>fileno()</td>
</tr>
<tr>
<td>creat()</td>
<td>flink()</td>
</tr>
<tr>
<td>creat64()</td>
<td>flock()</td>
</tr>
<tr>
<td>ctime()</td>
<td>fnmatch()</td>
</tr>
<tr>
<td>ctime_r()</td>
<td>fork()</td>
</tr>
<tr>
<td>daemon()</td>
<td>forkpty()</td>
</tr>
<tr>
<td>delay()</td>
<td>fpathconf()</td>
</tr>
<tr>
<td>devctl()</td>
<td>fseek()</td>
</tr>
<tr>
<td>dirname()</td>
<td>fseeko()</td>
</tr>
<tr>
<td>dispatch_block()</td>
<td>fnmatch()</td>
</tr>
<tr>
<td>dispatch_unblock()</td>
<td>fork()</td>
</tr>
<tr>
<td>div()</td>
<td>fsetpos()</td>
</tr>
<tr>
<td>dn_comp()</td>
<td>fstat()</td>
</tr>
<tr>
<td>dn_expand()</td>
<td>fstatvfs()</td>
</tr>
<tr>
<td>ds_clear()</td>
<td>fstatvfs64()</td>
</tr>
<tr>
<td>ds_create()</td>
<td>fstat64()</td>
</tr>
<tr>
<td>ds_deregister()</td>
<td>ftruncate()</td>
</tr>
<tr>
<td>ds_flags()</td>
<td>ftruncate64()</td>
</tr>
<tr>
<td>ds_set()</td>
<td>ftrylockfile()</td>
</tr>
</tbody>
</table>
Signal handlers

ftw()

hwi_tag2off()

ftw64()
in16()

futime()
in16s()

fwide()
in32()

gai_strerror()
in32s()

getdomainname()
in8()

getdtablesize()
in8s()

getegid()
inbe16()

geteuid()
inbe32()

getgid()
index()

getgroupslist()
inet6_option_alloc()

getgroups()
inet6_option_append()

gethostname()
inet6_option_find()

getitimer()
inet6_option_init()

getpgid()
inet6_option_next()

getpgrp()
inet6_option_space()

getpid()
inet6_rhdr_add()

getppid()
inet6_rhdr_gettime()

getprio()
inet6_rhdr_getflags()

getrlimit()
inet6_rhdr_init()

getrlimit64()
inet6_rhdr_lasthop()

getrusage()
inet6_rhdr_reverse()

getsubopt()
inet6_rhdr_segments()

gmtimeofday()
inet6_rhdr_space()

getuid()
inet_getaddr()

getw()
inet_aton()

getwd()
inet_inaof()

glob()
inet_makeaddr()

globfree()
inet_netof()

gmtime_r()
inet_network()

hsearch()
inet_ntop()

hstrerror()
inet_nton()

htonl()
inle16()

htons()
inle32()

hwi_find_item()
iofdinfo()

hwi_find_tag()
iofunc_attr_init()

hwi_off2tag()
iofunc_attr_lock()
Signal handlers

iofunc_attr_trylock()  iofunc_openfd()
iofunc_attr_unlink()  iofunc_openfd_default()
iofunc_check_access()  iofunc_pathconf()
iofunc_chmod()  iofunc_chmod_default()
iofunc_chmod_default()  iofunc_read_default()
iofunc_chown()  iofunc_read_verify()
iofunc_chown_default()  iofunc_readlink()
iofunc_client_info()  iofunc_rename()
iofunc_close_dup()  iofunc_space_verify()
iofunc_close_dup_default()  iofunc_stat()
iofunc_close_ocb()  iofunc_stat_default()
iofunc_close_ocb_default()  iofunc_sync()
iofunc_devctl()  iofunc_sync_default()
iofunc_devctl_default()  iofunc_sync_verify()
iofunc_fdinfo()  iofunc_time_update()
iofunc_fdinfo_default()  iofunc_unblock()
iofunc_func_init()  iofunc_unblock_default()
iofunc_link()  iofunc_unlink()
iofunc_lock()  iofunc_unlock_ocb_default()
iofunc_lockcalloc()  iofunc_utime()
iofunc_lock_default()  iofunc_utime_default()
iofunc_lock_free()  iofunc_write_default()
iofunc_lock_ocb_default()  ionotify()
iofunc_lseek()  ipsec_get_policylen()
iofunc_lseek_default()  ipsec_strerror()
iofunc_mknod()  isalnum()
iofunc_mmap()  isalpha()
iofunc_mmap_default()  isascii()
iofunc_notify()  iscntrl()
iofunc_notify_remove()  isdigit()
iofunc_notify_trigger()  isdigit()
iofunc_ocb_attach()  isgraph()
iofunc_ocb_calloc()  islower()
iofunc_ocb_detach()  isprint()
iofunc_ocb_free()  ispunct()
iofunc_open()  isspace()
iofunc_open_default()  isupper()
### Signal handlers

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iswalnum()</code></td>
<td><code>mbsinit()</code></td>
</tr>
<tr>
<td><code>iswalphal()</code></td>
<td><code>mbstrtwocs()</code></td>
</tr>
<tr>
<td><code>iswcntrl()</code></td>
<td><code>mbstowcs()</code></td>
</tr>
<tr>
<td><code>iswcctype()</code></td>
<td><code>mbtowc()</code></td>
</tr>
<tr>
<td><code>iswdigit()</code></td>
<td><code>mem_offset()</code></td>
</tr>
<tr>
<td><code>iswgraph()</code></td>
<td><code>mem_offset64()</code></td>
</tr>
<tr>
<td><code>iswlower()</code></td>
<td><code>memalign()</code></td>
</tr>
<tr>
<td><code>iswprint()</code></td>
<td><code>memccpy()</code></td>
</tr>
<tr>
<td><code>iswpunct()</code></td>
<td><code>memcmp()</code></td>
</tr>
<tr>
<td><code>iswspace()</code></td>
<td><code>memchr()</code></td>
</tr>
<tr>
<td><code>iswupper()</code></td>
<td><code>memcmp()</code></td>
</tr>
<tr>
<td><code>iswxdigit()</code></td>
<td><code>memcpy()</code></td>
</tr>
<tr>
<td><code>isxdigit()</code></td>
<td><code>memncpy()</code></td>
</tr>
<tr>
<td><code>itoa()</code></td>
<td><code>memmove()</code></td>
</tr>
<tr>
<td><code>kill()</code></td>
<td><code>memset()</code></td>
</tr>
<tr>
<td><code>killpg()</code></td>
<td><code>min()</code></td>
</tr>
<tr>
<td><code>labs()</code></td>
<td><code>mkdir()</code></td>
</tr>
<tr>
<td><code>lchown()</code></td>
<td><code>mkfifo()</code></td>
</tr>
<tr>
<td><code>ldiv()</code></td>
<td><code>mktemp()</code></td>
</tr>
<tr>
<td><code>lfind()</code></td>
<td><code>mktemp()</code></td>
</tr>
<tr>
<td><code>link()</code></td>
<td><code>mktime()</code></td>
</tr>
<tr>
<td><code>lio_listio()</code></td>
<td><code>mmap()</code></td>
</tr>
<tr>
<td><code>lltoa()</code></td>
<td><code>mmap64()</code></td>
</tr>
<tr>
<td><code>localtime_r()</code></td>
<td><code>mmap_device_io()</code></td>
</tr>
<tr>
<td><code>lockf()</code></td>
<td><code>mmap_device_memory()</code></td>
</tr>
<tr>
<td><code>login_tty()</code></td>
<td><code>modem_open()</code></td>
</tr>
<tr>
<td><code>longjmp()</code></td>
<td><code>modem_write()</code></td>
</tr>
<tr>
<td><code>lsearch()</code></td>
<td><code>mount()</code></td>
</tr>
<tr>
<td><code>lseek()</code></td>
<td><code>mprotect()</code></td>
</tr>
<tr>
<td><code>lseek64()</code></td>
<td><code>mq_timedreceive()</code></td>
</tr>
<tr>
<td><code>lstat()</code></td>
<td><code>mq_timedsend()</code></td>
</tr>
<tr>
<td><code>lstat64()</code></td>
<td><code>msync()</code></td>
</tr>
<tr>
<td><code>ltoa()</code></td>
<td><code>munmap()</code></td>
</tr>
<tr>
<td><code>max()</code></td>
<td><code>munmap_device_io()</code></td>
</tr>
<tr>
<td><code>mblen()</code></td>
<td><code>munmap_device_memory()</code></td>
</tr>
<tr>
<td><code>mbrlen()</code></td>
<td><code>name_close()</code></td>
</tr>
</tbody>
</table>
Signal handlers

name_open()  pathmgr_unlink()
nanospin()  pause()
nanospin_calibrate()  pccard_arm()
nanospin_count()  pccard_attach()
nanospin_ns()  pccard_detach()
nanospin_ns_to_count()  pccard_info()
nap()  pccard_lock()
napms()  pccard_raw_read()
ndacsteller_result()  pccard_unlock()
netmgr_netiostr()  pci_attach()
netmgr_remote_rd()  pci_attach_device()
netmgr_strtond()  pci_detach()
ntfw()  pci_detach_device()
ntfw64()  pci_find_class()
nice()  pci_find_device()
nsec2timespec()  pci_irq_routing_options()
ntohl()  pci_map_irq()
ntohs()  pci_present()
offsetof()  pci_read_config()
open()  pci_read_config16()
open64()  pci_read_config32()
openfd()  pci_read_config8()
openpty()  pci_rescan_bus()
out16()  pci_write_config()
out16s()  pci_write_config16()
out32()  pci_write_config32()
out32s()  pci_write_config8()
out8()  pipe()
out8s()  posix_mem_offset()
outbe16()  posix_mem_offset64()
outbe32()  posix_memalign()
outle16()  pread()
outle32()  pread64()
pathconf()  procmgr_daemon()
pathfind()  procmgr_event_notify()
pathfind_r()  procmgr_event_trigger()
pathmgr_symlink()  procmgr_guardian()
Signal handlers

pthread_abort()
pthread_atfork()
pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_getguardsize()
pthread_attr_getinheritsched()
pthread_attr_getschedparam()
pthread_attr_getschedpolicy()
pthread_attr_getscope()
pthread_attr_getstackaddr()
pthread_attr_getstacklazy()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_setguardsize()
pthread_attr_setinheritsched()
pthread_attr_setschedparam()
pthread_attr_setschedpolicy()
pthread_attr_setscope()
pthread_attr_setstackaddr()
pthread_attr_setstacklazy()
pthread_attr_setstacksize()
pthread_barrier_destroy()
pthread_barrier_init()
pthread_barrier_wait()
pthread_barrierattr_destroy()
pthread_barrierattr_getpshared()
pthread_barrierattr_init()
pthread_barrierattr_setpshared()
pthread_cancel()
pthread_cleanup_pop()
pthread_cleanup_push()
pthread_cond_broadcast()
pthread_cond_destroy()
pthread_cond_init()
pthread_cond_signal()
pthread_cond_timedwait()
Signal handlers

pthread_mutexattr_settype()  qnx_crypt()
pthread_once()  raise()
pthread_rwlock_destroy()  rand()
pthread_rwlock_init()  rand_r()
pthread_rwlock_rdlock()  random()
pthread_rwlock_timedrdlock()  rdchk()
pthread_rwlock_timedwrlock()  re_compt()
pthread_rwlock_tryrdlock()  re_exec()
pthread_rwlock_trywrlock()  read()
pthread_rwlock_unlock()  readblock()
pthread_rwlock_wrlock()  readcond()
pthread_rwlockattr_destroy()  readdir_r()
pthread_rwlockattr_getpshared()  readlink()
pthread_rwlockattr_init()  readv()
pthread_rwlockattr_setpshared()  realpath()
pthread_self()  regerror()
pthread_setcancelstate()  rename()
pthread_setsconcantype()  resmgr_msgread()
pthread_setsconcurrency()  resmgr_msgreadv()
pthread_setschedparam()  resmgr_msgwrite()
pthread_sigmask()  resmgr_msgwritev()
pthread_sleepon_broadcast()  resmgr_pathname()
pthread_sleepon_lock()  rewinddir()
pthread_sleepon_signal()  rindex()
pthread_sleepon_timedwait()  rmdir()
pthread_sleepon_unlock()  rsrdbmgr_attach()
pthread_sleepon_wait()  rsrdbmgr_create()
pthread_spin_destroy()  rsrdbmgr_destroy()
pthread_spin_init()  rsrdbmgr_detach()
pthread_spin_lock()  rsrdbmgr_devno_attach()
pthread_spin_trylock()  rsrdbmgr_devno_detach()
pthread_spin_unlock()  rsrdbmgr_query()
pthread_testcancel()  scandir()
pthread_timedjoin()  sched_get_priority_adjust()
putw()  sched_get_priority_max()
pwrite()  sched_get_priority_min()
pwrite64()  sched_getparam()
sched_getscheduler()  sigaddset()
sched_rr_get_interval()  sigblock()
sched_setscheduler()  sigdelset()
sched_yield()  sigemptyset()
sem_close()  sigfillset()
sem_destroy()  sigismember()
sem_getvalue()  siglongjmp()
sem_open()  sigmask()
sem_post()  signal()
sem_timedwait()  sigpause()
sem_trywait()  sigpending()
sem_unlink()  sigprocmask()
sem_wait()  sigqueue()
setdomainname()  sigsetjmp()
setegid()  sigsetmask()
seteuid()  sigsuspend()
setgid()  sigtimedwait()
sethostname()  sigunblock()
setitimer()  sigwait()
setjmp()  sigwaitinfo()
setpgid()  sleep()
setpgid()  slobg()
setpgrp()  slogf()
setpriority()  slogi()
setregid()  sprintf()
setreuid()  sopenfd()
setrlimit()  spawn()
setrlimit64()  sprintf()
setsid()  srand()
settimeofday()  srand48()
setuid()  sscanf()
setutent()  stat()
shm_ctl()  stat64()
shm_ctl_special()  statvfs()
shm_open()  statvfs64()
shm_unlink()  straddstr()
sigaction()  strcasecmp()
Signal handlers

`strcat()`  `swprintf()`
`strchr()`  `swscanf()`
`strcmp()`  `symlink()`
`strcmpi()`  `sync()`
`strcoll()`  `sysconf()`
`strcpy()`  `sysmgr_reboot()`
`strcspn()`  `tcdrain()`
`strerror()`  `tcdropline()`
`strftime()`  `teflow()`
`stricmp()`  `tcflush()`
`strlen()`  `tcgetattr()`
`strlwr()`  `tcgetattrr()`
`strncasecmp()`  `tcgetpgrp()`
`strncat()`  `tcgetsid()`
`strncmp()`  `tcgetsize()`
`strncpy()`  `tcinjection()`
`strnicmp()`  `tciscars()`
`strncpy()`  `tcsendbreak()`
`strpbrk()`  `tcsetattr()`
`strrchr()`  `tcsetattrattr()`
`strrev()`  `tell()`
`strsep()`  `tell64()`
`strset()`  `time()`
`strsignal()`  `timer_create()`
`strspn()`  `timer_delete()`
`strsbrk()`  `timer_getexpstatus()`
`strtok()`  `timer_gettime()`
`strtol()`  `timer_getoverrun()`
`strtoimax()`  `timer_gettime()`
`strtoull()`  `timer_settime()`
`strtooll()`  `timer_timeout()`
`strtoomax()`  `timer_timeout_r()`
`strtoomax()`  `times()`
`strtoomax()`  `timespec2nsec()`
`strtoomax()`  `tolower()`
`strtoomax()`  `toupper()`
`strtoomax()`  `towctrans()`
`swab()`  `tolower()`
towupper()  wcscpy()
truncate()  wcscspn()
ttynamex_r()  wcsxfmt()
ualarm()  wcftime()
ulioa()  wcslen()
ultoa()  wcscncat()
umask()  wcscmp()
uname()  wcscpy()
unlink()  wcespbrk()
unsetenv()  wcsrchr()
usleep()  wcsrtombs()
utime()  wcsspn()
utimes()  wcsstr()
utmpname()  wcstod()
utoa()  wcstof()
va_arg()  wcstomax()
va_copy()  wcstok()
va_end()  wcstol()
va_start()  wcstold()
valloc()  wcstoll()
verr()  wcstombs()
verrx()  wcstoul()
vslogf()  wcstoull()
vwarn()  wcstoumax()
vwarnx()  wctob()
wait()  wctomb()
wait3()  wctrans()
wait4()  wctype()
waitid()  wmemchr()
waitpid()  wmemcmp()
warn()  wmemcpy()
warnx()  wmemmove()
wctomb()  wmemset()
wcsnctou()  wordexp()
wcschr()  wordfree()
wcsncmp()  write()
wcsncpy()  writeblock()
writev()

See the “Caveats” section for the following functions for more information:

- abort()
- modem_read()
- modem_script()
- vsnprintf()
- vsprintf()
- vsscanf()
- vswprintf()
- vswscanf()

Multithreaded programs

CAUTION: It isn’t safe to call these functions from a multithreaded program.

- Raccept()
- Rbind()
- Rconnect()
- Rgetsockname()
- Rlisten()
- Rrcmd()
- Rselect()
- SOCKSinit()
- bindresvport()
- crypt()
- daemon()
- drand48()
- endgrent()
- endhostent()
- endnetent()
- endprotoent()
- endpwent()
- endspent()
- endutent()
- fgetspent()
getpwent()  
getpwnam()  
getpwuid()  
getservbyname()  
getservbyport()  
getservent()  
gmtime()  
herror()  
inet_ntoa()  
initgroups()  
initstate()  
input_line()  
ioctl()  
lcong48()  
localtime()  
lrand48()  
lstat()  
lstat64()  
rand48()  
pclose()  
popen()  
putc_unlocked()  
putchar_unlocked()  
putenv()  
putspent()  
rand()  
random()  
rcmd()  
read_main_config_file()  
readdir()  
res_init()  
res_mkquery()  
res_query()  
res_querydomain()  
res_search()  
res_send()  
ruserok()  
seekdir()  
setenv()  
setgrent()  
setgroups()  
sethostent()  
setpwent()  
setservent()  
setstate()  
sigblock()  
snmp_close()  
snmp_free_pdu()  
snmp_open()  
snmp_pdu_create()  
snmp_read()  
snmp_select_info()  
snmp_send()  
snmp_timeout()  
socketatmark()  
srandom()  
strtok()  
syslog()  
telldir()  
tempnam()  
ttyname()  
vfork()  
vsyslog()  
wcscoll()
See the “Caveats” section for the following functions for more information:

- `ctermid()`
- `modem_read()`
- `modem_script()`
- `select()`
- `tmpnam()`
Appendix C

What’s New in this Reference?

In this appendix...

What’s new in QNX Momentics 6.3.2? 3845
What’s new in the QNX Neutrino Core OS 6.3.2? 3845
What’s new in QNX Neutrino 6.3.0 Service Pack 2? 3847
What’s new in QNX Neutrino 6.3.0 Service Pack 1? 3849
What’s new in QNX Neutrino 6.3.0? 3852
What’s new in QNX Neutrino 6.2.1? 3854
What’s new in QNX Neutrino 6.2? 3857
What’s new in the QNX Neutrino 6.1.0 docs? 3861
What’s new in QNX Momentics 6.3.2?

Errata

`clock()`  This function returns the number of clock ticks, or `(clock_t) -1` if the number of ticks couldn’t be determined or exceeds the maximum value that the `clock_t` type can represent.

`GETIOVBASE(), GETIOVLLEN()`

The argument to these macros is a pointer to an `iov_t` structure.

What’s new in the QNX Neutrino Core OS 6.3.2?

Significant changes:

- New content
- Changed content

New Content

`posix_TYPED_mem_get_info(), posix_TYPED_mem_open()`

New functions for POSIX support of memory-mapping.

`pthread_getname_np(), pthread_setname_np()`

The `pthread_getname_np()` function retrieves the name of a thread. The `pthread_setname_np()` function names a thread in your application.

`pthread_attr_getstackprealloc(), pthread_attr_setstackprealloc()`

Get and set the memory to preallocate for a MAP_LAZY stack.
munmap_flags()

New function to unmap previously mapped addresses exercising more control.

SchedCtl()  Control the adaptive partitioning scheduler

SIGEV_CLEAR_CRITICAL()

In adaptive partitioning, make a thread run as noncritical. See sigevent.

SIGEV_GET_TYPE()

Determine the type of a sigevent event. See sigevent.

SIGEV_MAKE_CRITICAL()

In adaptive partitioning, make a thread run as critical. See sigevent.

Changed Content

ChannelCreate()

If you’re using adaptive partitioning and you set _NTO_CHF_FIXED_PRIORITY, the receiving thread won’t run in the sending thread’s partition.

getrlimit(), getrlimit64(), setrlimit(), setrlimit64()

Added the following resources:

- RLIMIT_MEMLOCK
- RLIMIT_NPROC
- RLIMIT_NTHR
- RLIMIT_OFILE
- RLIMIT_RSS

iofunc_open_default()

The documentation now describes the extra argument.
mlock(), mlockall(), munlock()

The documentation now describes the POSIX support for memory mapping.

mmap()

The documentation now includes a new MAP_NOINIT flag.

msync()

The documentation now describes new MS_CACHE_ONLY flag.

posix_mem_offset(), posix_mem_offset64()

The documentation now describes the POSIX support for memory-mapping.

sigevent

In the future, the library might use hidden bits in the sigev_notify member of the sigevent structure. Don’t compare this member directly to values such as SIGEV_PULSE; use the SIGEV_GET_TYPE() macro.

spawn()

The documentation now describes the SPAWN_EXPLICIT_CPU flag and the runmask member of the inheritance structure.

ThreadCtl()

The documentation now contains the following commands:

- _NTO_TCTL_NAME
- _NTO_TCTL_ONE_THREAD_HOLD
- _NTO_TCTL_ONE_THREAD_CONT
- _NTO_TCTL_RUNMASK_GET_AND_SET
- _NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT
New Content

\texttt{cache\_fini()} Free cache-coherency resources when the driver is unloaded.

\texttt{CACHE\_FLUSH()}
Flush cache line associated to a data buffer.

\texttt{cache\_init()} Register with the cache coherency library.

\texttt{CACHE\_INVAL()}
Invalidates cache line associated to a data buffer.

Changed Content

\texttt{mq\_close()}, \texttt{mq\_getattr()}, \texttt{mq\_notify()}, \texttt{mq\_open()}, \texttt{mq\_receive()}, \texttt{mq\_send()}, \texttt{mq\_setattr()}, \texttt{mq\_timedreceive()}, \texttt{mq\_timedsend()}, \texttt{mq\_unlink()}
Added information about the traditional (\texttt{mqueue}) and alternate (\texttt{mq}) implementations of message queues.

\texttt{nanospin()}, \texttt{nanospin\_calibrate()}, \texttt{nanospin\_count()}, \texttt{nanospin\_ns()}, \texttt{nanospin\_ns\_to\_count()}
The \texttt{nanospin\_*() functions are designed for use with hardware that requires short time delays between accesses. You should use them to delay only for times less than a few milliseconds. For longer delays, use the POSIX \texttt{timer\_\_*()} functions.

\texttt{pci\_attach\_device()}, \texttt{pci\_find\_class()}, \texttt{pci\_find\_device()}
For a list of supported device and vendor IDs, see \texttt{<hw/pci\_devices.h>}; for a list of class and subclass codes, see \texttt{<hw/pci.h>}.

Errata

\_\texttt{cmdname()} Corrected the information about what this function returns.

\texttt{openlog()} Corrected the name of the \texttt{LOG\_FTP} facility.
pci_attach_device()

The BusNumber and DevFunc members of the pci_dev_info structure are input/output.

pci_irq_routing_options()

This function is for x86 only.

sem_close(), sem_open(), sem_unlink()

Named semaphores are now managed by procnto, not mqueue.

thread_pool_create()

The description of the block_func member of the thread_pool_attr_t structure has been corrected.

vfprintf(), vprintf()

The code samples now compile without warnings.

What’s new in QNX Neutrino 6.3.0 Service Pack 1?

New content

asyncmsg_channel_create()

Create an asynchronous message channel.

asyncmsg_channel_destroy()

Destroy an asynchronous message channel.

asyncmsg_connect_attach()

Establish a connection between a process and a channel.

asyncmsg_connect_attr()

Return the original connection attributes.

asyncmsg_connect_detach()

Break a connection between a process and a channel.
asyncmsg_flush()
Flush the messages sent through the connection.

asyncmsg_free()
Free a message buffer.

asyncmsg_get()
Receive an asynchronous message.

asyncmsg_malloc()
Allocate a message buffer for sending.

asyncmsg_put(), asyncmsg_putv()
Send asynchronous messages to a connection.

shm_ctl_special()
Give special attributes to a shared memory object

**Changed content**

getdomainname()  
If the buffer isn’t large enough, `getdomainname()` truncates the domain name.

groupulist() The Neutrino implementation of this function ignores the basegid argument.

mallopt() The MALLOC_CKACCESS, MALLOC_FILLAREA, and MALLOC_CKCHAIN options were added to this call.

`MsgReply()`, `MsgReply_r()`, `MsgReplyv()`, `MsgReplyv_r()`  
The `MsgSend*_r()` functions use negative `errno` values to indicate failure, so you shouldn’t pass a negative value for the status to `MsgReply*_()`, because the `MsgSend*_r()` functions could interpret it as an error code.
name_attach()  The example now handles an _IO_CONNECT message.

nanospin(), nanospin_ns(), and nanospin_ns_to_count()  The first time that you call these functions, the C library invokes nanospin_calibrate() with an argument of 0 (interrupts enabled), unless you call it directly first.

pci_attach_device()  Added PCI_MASTER_ENABLE to the flags.

procmgr_daemon()  The data in the siginfo_t structure for the SIGCHLD signal that the parent receives isn’t useful in this case.

rsrcdbmgr_create()  Added RSRCDBMGR_FLAG_NOREMOVE to the flags.

setenv()  This function doesn’t free any memory. If you want to change the value of an existing environment variable, you should use putenv() instead.

sigevent  If you don’t want to modify the priority of the thread that receives the pulse, specify SIGEV_PULSE_Prio_INHERIT for the priority when you call SIGEV_PULSE_INIT().

spawn(), spawnp()  
- Added descriptions of the rest of the flags for the inheritance structure.
- If you set SPAWN_EXEC in the flags member of the inheritance structure, these functions don’t return, unless an error occurred.
Errata

getsubopt()  Corrected the example.

mq_notify()  Don’t use SIGEV_INTR as an event type.

pause()  This function suspends the calling thread, not the process, until delivery of a signal.

pthread_setschedparam()

- The timeslice for round-robin scheduling (SCHED_RR) is $4 \times$ the clock period.
- You can specify sporadic scheduling at any time, not just when you create a thread.

regexec()  Corrected the description of the regmatch_t structure.

sched_get_priority_max(), sched_get_priority_min(),
sched_setscheduler(), SchedInfo(), SchedSet()

The timeslice for round-robin scheduling (SCHED_RR) is $4 \times$ the clock period.

sched_setscheduler(), SchedSet()

You can specify sporadic scheduling at any time, not just when you create a thread.

spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(),
spawnv(), spawnve(), spawnvp(), spawnvpe()

The child process can’t access the parent process’s environment, only its own.

What’s new in QNX Neutrino 6.3.0?

New content

fopen64()  Large-file support for fopen().

freopen64()  Large-file support for freopen().

ftw64()  Large-file support for ftw().
What’s New in QNX Neutrino 6.3.0?

- `getlineinfo()` Perform address-to-nodename translation.
- `inet6_option_*()` Manipulate IPv6 hop-by-hop and destination options.
- `inet6_rthdr_*()` manipulate IPv6 Router header options.
- `ipsec_dump_policy()` Generate a readable string from an IPsec policy specification.
- `ipsec_get_policylen()` Get the length of the IPsec policy.
- `ipsec_set_policy()` Generate an IPsec policy specification structure from a readable string.
- `nftw(), nftw64()` Walk a file tree and its large-file support.
- `poll()` Input/output multiplexing.
- `resmgr_handle_tune()` Tune aspects of client fd-to-OCB mapping.
- `sctp_bindx()` Add or remove one or more addresses from a given association.
- `sctp_connectx()` Help associate an endpoint that is multi-homed.
- `sctp_freeladdr()` Free all resources allocated by `sctp_getladdr()`.
- `sctp_freepaddr()` Free all resources allocated by `sctp_getpaddr()`.
- `sctp_getladdr()` Return all locally bound addresses on a socket.
What's new in QNX Neutrino 6.2.1?

New content

dispatch_unblock()

Unblock all of the threads that are blocked on a dispatch handle.

erno

Each thread in a multi-threaded program has its own error value in its thread local storage. No matter which thread you’re in, you can simply refer to errno — it’s defined in such a way that it refers to the correct variable for the thread. For more information, see “Local storage for private data” in the documentation for ThreadCreate().

pthread_attr_setschedpolicy().

Sporadic scheduling (SCHED_SPORADIC) is a new feature of QNX Neutrino 6.2.0.

sched_param

Structure of scheduling parameters

va_copy() Make a copy of a variable argument list

sctp_getpaddr() Return all peer addresses in an association.

sctp_peeloff() Branch off an association into a separate socket.

SCTP Stream Control Transmission Protocol.

sctp_recmsg() Receive message using advanced SCTP features.

sctp_sendmsg() Send message using advanced SCTP features.

tmpfile64() Large-file support for tmpfile().
Changed content

`bind()`, `bindresvport()`

These functions aren’t cancellation points any more, because this conflicted with POSIX.

`htonl()`, `htons()`, `inet_ntop()`, `inet_ntpton()`, `isfdtype()`, `ntohl()`, `ntohs()`

These functions have been moved from `libsocket` to `libc`.

Errata

`alphasort()` This function compares two directory entries; it doesn’t sort an array of entries.

`execlpe()`, `execve()`

You can now execute a shell script.

`fgetc()`, `fgetchar()`, `fgets()`, `fgetwc()`, `fgetws()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, `gets()`, `getw()`, `getwc()`, `getwchar()`

Use `feof()` or `ferror()` to distinguish an end-of-file condition from an error.

`fstat()`, `fstat64()` These functions return -1 if an error occurs.

`iofunc_mmap()`, `iofunc_mmap_default()` These functions return a nonpositive value on success.

`InterruptAttach()`, `InterruptAttachEvent()` You should always set `_NTO_INTR_FLAGS_TRK_MSK`.

`mq_getattr()`, `mq_setattr()` The `mq_flags` member of the `mq_attr` structure applies to the message-queue description (i.e. locally), not to the queue as a whole.

`mq_open()` Corrected the interpretation of the `name` argument.
MsgError(), MsgError_r()

If the error argument is EOK, the MsgSend*() call returns EOK; if error is any other value, the MsgSend*() call returns -1.

MsgSendPulse(), MsgSendPulse_r()

You can now send pulses across the network. You can send a pulse to any process — not just to a process in the same process group — if your process has the appropriate permission.

name_open()

This function returns a nonnegative integer representing a side-channel connection ID, or -1 if an error occurred.

printf()

The exponent produced for the e and E formats is at least two digits long.

Clarified what happens if the format string includes invalid multibyte characters.

pthread_mutex_timedlock(), pthread_rwlock_timedrdlock(), pthread_rwlock_timedwrlock()

The timeout is based on the CLOCK_REALTIME clock.

_resmgr_ocb()

Corrected the name.

select()

This function and the associated macros are now defined in <sys/select.h>, instead of <sys/time.h> (which includes <sys/select.h>).

sem_open()

Corrected the interpretation of the sem_name argument.

sem_timedwait()

The timeout is based on the CLOCK_REALTIME clock.

send()

The list of errors now includes EPIPE.
What's new in QNX Neutrino 6.2?

Significant changes:

- New content
- Deprecated content
- Errata

New Content

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addrinfo</td>
<td>TCP/IP address information</td>
</tr>
<tr>
<td>dircntl()</td>
<td>Control an open directory</td>
</tr>
<tr>
<td>freeaddrinfo()</td>
<td>Free an address information structure</td>
</tr>
<tr>
<td>freeifaddrs()</td>
<td>Free an address information structure</td>
</tr>
<tr>
<td>gai_strerror()</td>
<td>Return the getaddrinfo() error code</td>
</tr>
<tr>
<td>getaddrinfo()</td>
<td>Get address information</td>
</tr>
<tr>
<td>getdomainname()</td>
<td>Get the domain name of the current host</td>
</tr>
</tbody>
</table>

You can now execute a shell script. The child process’s tms_utime, tms_stime, tms_cutime, and tms_cstime are now calculated.
gethostbyname2() Get a network host entry, given a name
getifaddrs() Get a network interface address
hwi_find_item() Find an item in the hwi_item structure
hwi_find_tag() Find a tag in the hwi_item structure
hwi_off2tag() Return a pointer to the start of a tag in the hwinfo area of the system page
hwi_tag2off() Return the offset from the start of the hwinfo area of the system page
ICMP6 Internet Control Message Protocol for IPv6
if_freenameindex() Free dynamic memory allocated by if_nameindex()
if_indextoname() Map an interface index to its name
if_nameindex() Return a list of interfaces
if_nametoindex() Map an interface name to its index
ifaddrs() Structure that describes an Internet host
INET6 Internet Protocol version 6 family
inet_net_ntop() Convert an Internet network number to CIDR format
inet_net_pton() Convert an Internet network number from CIDR format to network format
IPv6 Internet Protocol version 6
IPsec Internet security protocol
mallinfo()    Get memory allocation information
mallopt()     Control the memory allocation
mcheck()      Enable memory allocation routine consistency checks
memalign()    Allocate aligned memory
mprobe()      Perform consistency check on memory
posix_memalign()    Allocate aligned memory
procmgr_session()  Provide process manager session support
_resmgr_handle_grow()  Expand the capacity of the device manager database
_resmgr_io_func()    Retrieve an I/O function from an I/O function table
resmgr_iofuncs()    Extract the I/O function pointers associated with connection
_resmgr_ocb()     Retrieve an Open Control Block
sched_get_priority_adjust()  Calculate the allowable priority for the scheduling policy
seekdir()       Set the position for the next read of the directory stream
_sleepon_broadcast()  Wake up multiple threads
What's new in QNX Neutrino 6.2?

© 2007, QNX Software Systems GmbH & Co. KG.

```c
_sleepon_destroy()
    Destroy a sleepon
_sleepon_init()    Initialize a sleepon
_sleepon_lock()    Lock a sleepon
_sleepon_signal()
    Wake up a single thread
_sleepon_unlock()
    Unlock a sleepon
_sleepon_wait()    Wait on a sleepon
tcsetsid()
    Make a terminal device a controlling device
strtoimax(), strtoumax()
    Convert a string to an integer type
telldir()
    Get the location associated with the directory stream
valloc()
    Allocate a heap block aligned on a page boundary
wcstoimax(), wcstoumax()
    Convert a wide-character string to an integer type
```

**Deprecated Content**

- `getpriority()` — use `getprio()` or `SchedGet()` instead.
- `setpriority()` — use `setprio()` or `SchedSet()` instead.

**Errata**

`snprintf()` — Corrected the Returns section and Classifications
What’s new in the QNX Neutrino 6.1.0 docs?

Significant changes:
- New content
- Deprecated content

New content

The following functions have been added:

Wide-character functions
- Wide-character versions of many functions

*InterruptHookTrace()*
  - Attach the pseudo interrupt handler that’s used by the instrumented module

*iofdinfo()*
  - Retrieve server attributes

*iofunc_fdinfo()*
  - Handle an _IO_FDINFO message

*iofunc_fdinfo_default()*
  - Default handler for _IO_FDINFO messages

*MsgVerifyEvent(), MsgVerifyEvent_r()*
  - Check the validity of a receive ID and an event configuration

*resmgr_unbind()*
  - Remove an OCB

*straddstr()*
  - Concatenate one string on to the end of another

*SyncCtl(), SyncCtl_r()*
  - Perform an operation on a synchronization object

*SyncMutexEvent(), SyncMutexEvent_r()*
  - Attach an event to a mutex
What's new in the QNX Neutrino 6.1.0 docs? © 2007, QNX Software Systems GmbH & Co. KG.

SyncMutexRevive(), SyncMutexRevive_r()
   Revive a mutex

thread_pool_control()
   Control the thread pool behavior

thread_pool_limits()
   Wrapper function for thread_pool_control()

TraceEvent()   Trace kernel events

Deprecated content

matherr()   Handle errors in math library functions
Appendix D

Third-Party Copyright Notices
BSD stack

Copyright © 1997 Christopher G. Demetriou. All rights reserved.

Copyright © 1982, 1986, 1989, 1991, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

BSD stack and various utilities

Copyright © 1998 The NetBSD Foundation, Inc.
All rights reserved.
This code is derived from software contributed to The NetBSD Foundation by Public Access Networks Corporation ("Panix"). It was developed under contract to Panix by Eric Haszlakiewicz and Thor Lancelot Simon.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1 Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2 Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3 All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

4 Neither the name of The NetBSD Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright © 1995 The NetBSD Foundation, Inc. All rights reserved.

This code is derived from software contributed to The NetBSD Foundation by Christos Zoulas. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1 Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2 Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3 All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

4 Neither the name of The NetBSD Foundation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright © 1996, 1997 The NetBSD Foundation, Inc. All rights reserved.

This code is derived from software contributed to The NetBSD Foundation by Jason R. Thorpe of the Numerical Aerospace Simulation Facility, NASA Ames Research Center.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the NetBSD Foundation, Inc. and its contributors.
4. Neither the name of The NetBSD Foundation nor the names of its contributors may be used to endorse or promote products
derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright © 1996 Matt Thomas matt@3am-software.com. All rights reserved.

Copyright © 1982, 1986, 1988, 1993 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1  Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2  Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3  Neither the name of the University nor the names of its contributors may be used to endorse or promote products
derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions Copyright © 1993 by Digital Equipment Corporation.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of Digital Equipment Corporation not be used in advertising or publicity pertaining to distribution of the document or software without specific, written prior permission.

THE SOFTWARE IS PROVIDED “AS IS” AND DIGITAL EQUIPMENT CORP. DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL DIGITAL EQUIPMENT CORPORATION BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Portions Copyright © 1995 by International Business Machines, Inc.

International Business Machines, Inc. (hereinafter called IBM) grants permission under its copyrights to use, copy, modify, and distribute this Software with or without fee, provided that the above copyright notice and all paragraphs of this notice appear in all copies, and that the name of IBM not be used in connection with the marketing of any product incorporating the Software or modifications thereof, without specific, written prior permission.

To the extent it has a right to do so, IBM grants an immunity from suit under its patents, if any, for the use, sale or manufacture of products to the extent that such products are used for performing Domain Name System dynamic updates in TCP/IP networks by means of the Software. No immunity is granted for any product per se or for any other function of any product.

THE SOFTWARE IS PROVIDED “AS IS”, AND IBM DISCLAIMS ALL WARRANTIES, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE, EVEN IF IBM IS APPRISED OF THE POSSIBILITY OF SUCH DAMAGES.

Copyright © 1996 by Internet Software Consortium.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED “AS IS” AND INTERNET SOFTWARE CONSORTIUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

All of the documentation and software included in the third BSD Networking Software Release is copyrighted by The Regents of the University of California.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1 Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2 Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3 Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGligence OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

MINIX operating system

Copyright © 1987,1997
Prentice Hall
All rights reserved.

Redistribution and use of the MINIX operating system in source and
binary forms, with or without modification, are permitted provided
that the following conditions are met:

1 Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2 Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following
disclaimer in the documentation and/or other materials
provided with the distribution.

3 Neither the name of Prentice Hall nor the names of the software
authors or contributors may be used to endorse or promote
products derived from this software without specific prior
written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT
HOLDERS, AUTHORS, AND CONTRIBUTORS “AS IS” AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT
NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
PRENTICE HALL OR ANY AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

ncurses library

Copyright © 1998 Free Software Foundation, Inc.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, distribute with modifications, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE ABOVE COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name(s) of the above copyright holders shall not be used in advertising or otherwise to promote the
sale, use or other dealings in this Software without prior written authorization.

Regular-expression handling

Henry Spencer.
All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1 The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.

2 The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

3 Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

4 This notice may not be removed or altered.

Remote Procedure Call (RPC)

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, California 94043

Sun RPC is a product of Sun Microsystems, Inc. and is provided for unrestricted use provided that this legend is included on all tape media
and as a part of the software program in whole or part. Users may copy or modify Sun RPC without charge, but are not authorized to license or distribute it to anyone else except as part of a product or program developed by the user.

SUN RPC IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

Sun RPC is provided with no support and without any obligation on the part of Sun Microsystems, Inc. to assist in its use, correction, modification or enhancement.

SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY SUN RPC OR ANY PART THEREOF.

In no event will Sun Microsystems, Inc. be liable for any lost revenue or profits or other special, indirect and consequential damages, even if Sun has been advised of the possibility of such damages.

SNMPv2

Copyright © 1988, 1989, 1991
Carnegie Mellon University
All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Carnegie Mellon University not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

CARNEGIE MELLON UNIVERSITY DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL CMU BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

SOCKS

Copyright © 1989
The Regents of the University of California.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Portions Copyright © 1993, 1994 by NEC Systems Laboratory.

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and that the name of NEC Systems Laboratory not be used in advertising or publicity pertaining to distribution of the document or software without specific, written prior permission.

THE SOFTWARE IS PROVIDED “AS IS” AND NEC SYSTEMS LABORATORY DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL NEC SYSTEMS LABORATORY BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
**A20 gate**

On x86-based systems, a hardware component that forces the A20 address line on the bus to zero, regardless of the actual setting of the A20 address line on the processor. This component is in place to support legacy systems, but the QNX Neutrino OS doesn’t require any such hardware. Note that some processors, such as the 386EX, have the A20 gate hardware built right into the processor itself — our IPL will disable the A20 gate as soon as possible after startup.

**adaptive**

Scheduling algorithm whereby a thread’s priority is decayed by 1. See also FIFO, round robin, and sporadic.

**adaptive partitioning**

A method of dividing, in a flexible manner, CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

**asymmetric multiprocessing (AMP)**

A multiprocessing system where a separate OS, or a separate instantiation of the same OS, runs on each CPU.

**atomic**

Of or relating to atoms. :-)  
In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered indivisible. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread’s position.
attributes structure

Structure containing information used on a per-resource basis (as opposed to the OCB, which is used on a per-open basis).

This structure is also known as a handle. The structure definition is fixed (iofunc_attr_t), but may be extended. See also mount structure.

bank-switched

A term indicating that a certain memory component (usually the device holding an image) isn’t entirely addressable by the processor. In this case, a hardware component manifests a small portion (or “window”) of the device onto the processor’s address bus. Special commands have to be issued to the hardware to move the window to different locations in the device. See also linearly mapped.

base layer calls

Convenient set of library calls for writing resource managers. These calls all start with resmgr_*(). Note that while some base layer calls are unavoidable (e.g. resmgr_pathname_attach()), we recommend that you use the POSIX layer calls where possible.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an “extension” to the BIOS or ROM Monitor — control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes 0x55 and 0xAA must be present (in that order) as the first two bytes in the device, with control being transferred to offset 0x0003.

block-integral

The requirement that data be transferred such that individual structure components are transferred in their entirety — no partial structure component transfers are allowed.
In a resource manager, directory data must be returned to a client as block-integral data. This means that only complete struct dirent structures can be returned — it’s inappropriate to return partial structures, assuming that the next _IO_READ request will “pick up” where the previous one left off.

**bootable**

An image can be either bootable or nonbootable. A bootable image is one that contains the startup code that the IPL can transfer control to.

**bootfile**

The part of an OS image that runs the startup code and the Neutrino microkernel.

**bound multiprocessing (BMP)**

A multiprocessing system where a single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

**budget**

In sporadic scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

**buildfile**

A text file containing instructions for mkifs specifying the contents and other details of an image, or for mkefs specifying the contents and other details of an embedded filesystem image.

**canonical mode**

Also called edited mode or “cooked” mode. In this mode the character device library performs line-editing operations on each received character. Only when a line is “completely entered” — typically when a carriage return (CR) is received — will the line of data be made available to application processes. Contrast raw mode.
channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a **connection** (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using `ChannelCreate()`), and then receives messages from that channel (using `MsgReceive()`). Another thread that wishes to send a message to the first thread must make a connection to that channel by “attaching” to the channel (using `ConnectAttach()`) and then sending data (using `MsgSend()`).

CIFS

Common Internet File System (aka SMB) — a protocol that allows a client workstation to perform transparent file access over a network to a Windows 95/98/NT server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

CIS

Card Information Structure — a data block that maintains information about flash configuration. The CIS description includes the types of memory devices in the regions, the physical geometry of these devices, and the partitions located on the flash.

**combine message**

A resource manager message that consists of two or more messages. The messages are constructed as combine messages by the client’s C library (e.g. `stat()`, `readblock()`), and then handled as individual messages by the resource manager.

The purpose of combine messages is to conserve network bandwidth and/or to provide support for atomic operations. See also **connect message** and **I/O message**.
connect message

In a resource manager, a message issued by the client to perform an operation based on a pathname (e.g. an `io_open` message). Depending on the type of connect message sent, a context block (see OCB) may be associated with the request and will be passed to subsequent I/O messages. See also combine message and I/O message.

connection

A kernel object used with message passing.

Connections are created by client threads to “connect” to the channels made available by servers. Once connections are established, clients can `MsgSendv()` messages over them. If a number of threads in a process all attach to the same channel, then the one connection is shared among all the threads. Channels and connections are identified within a process by a small integer.

The key thing to note is that connections and file descriptors (FD) are one and the same object. See also channel and FD.

context

Information retained between invocations of functionality.

When using a resource manager, the client sets up an association or context within the resource manager by issuing an `open()` call and getting back a file descriptor. The resource manager is responsible for storing the information required by the context (see OCB). When the client issues further file-descriptor based messages, the resource manager uses the OCB to determine the context for interpretation of the client’s messages.

cooked mode

See canonical mode.
**core dump**

A file describing the state of a process that terminated abnormally.

**critical section**

A code passage that *must* be executed “serially” (i.e. by only one thread at a time). The simplest form of critical section enforcement is via a **mutex**.

**deadlock**

A condition in which one or more threads are unable to continue due to resource contention. A common form of deadlock can occur when one thread sends a message to another, while the other thread sends a message to the first. Both threads are now waiting for each other to reply to the message. Deadlock can be avoided by good design practices or massive kludges — we recommend the good design approach.

**device driver**

A process that allows the OS and application programs to make use of the underlying hardware in a generic way (e.g. a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn’t affect any other part of the OS — drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won’t cause the entire OS to shut down.

**discrete (or traditional) multiprocessor system**

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

**DNS**

Domain Name Service — an Internet protocol used to convert ASCII domain names into IP addresses. In QNX native networking, **dns** is one of **Qnet**’s builtin resolvers.
**dynamic bootfile**

An OS image built on the fly. Contrast static bootfile.

**dynamic linking**

The process whereby you link your modules in such a way that the Process Manager will link them to the library modules before your program runs. The word “dynamic” here means that the association between your program and the library modules that it uses is done at load time, not at linktime. Contrast static linking. See also runtime loading.

**edge-sensitive**

One of two ways in which a PIC (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is “noticed” upon a transition to/from the rising/falling edge of a pulse. Contrast level-sensitive.

**edited mode**

See canonical mode.

**EOI**

End Of Interrupt — a command that the OS sends to the PIC after processing all Interrupt Service Routines (ISR) for that particular interrupt source so that the PIC can reset the processor’s In Service Register. See also PIC and ISR.

**EPROM**

Erasable Programmable Read-Only Memory — a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g. 12V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. To change a bit from a 0 state into a 1 state can only be accomplished by erasing the entire device, setting all of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically
10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast flash and RAM.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requestor of the event.

FD

File Descriptor — a client must open a file descriptor to a resource manager via the open() function call. The file descriptor then serves as a handle for the client to use in subsequent messages. Note that a file descriptor is the exact same object as a connection ID (coid, returned by ConnectAttach()).

FIFO

First In First Out — a scheduling algorithm whereby a thread is able to consume CPU at its priority level without bounds. See also adaptive, round robin, and sporadic.

flash memory

A memory technology similar in characteristics to EPROM memory, with the exception that erasing is performed electrically instead of via ultraviolet light, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64k bytes at a time) instead of the entire device. Contrast EPROM and RAM.

FQNN

Fully Qualified NodeName — a unique name that identifies a QNX Neutrino node on a network. The FQNN consists of the nodename plus the node domain tacked together.
garbage collection
Aka space reclamation, the process whereby a filesystem manager recovers the space occupied by deleted files and directories.

HA
High Availability — in telecommunications and other industries, HA describes a system’s ability to remain up and running without interruption for extended periods of time.

handle
A pointer that the resource manager base library binds to the pathname registered via `resmgr_attach()`. This handle is typically used to associate some kind of per-device information. Note that if you use the `iofunc_*()` POSIX layer calls, you must use a particular type of handle — in this case called an attributes structure.

hard thread affinity
A user-specified binding of a thread to a set of processors, done by means of a `runmask`. Contrast soft thread affinity.

image
In the context of embedded QNX Neutrino systems, an “image” can mean either a structure that contains files (i.e. an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim FFS-2-compatible filesystem (i.e. a flash filesystem image).

inherit mask
A bitmask that specifies which processors a thread’s children can run on. Contrast `runmask`.

interrupt
An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.
interrupt handler

   See ISR.

interrupt latency

   The amount of elapsed time between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service routine. Also designated as “T_{i1}”. Contrast scheduling latency.

interrupt service routine

   See ISR.

interrupt service thread

   A thread that is responsible for performing thread-level servicing of an interrupt.

   Since an ISR can call only a very limited number of functions, and since the amount of time spent in an ISR should be kept to a minimum, generally the bulk of the interrupt servicing work should be done by a thread. The thread attaches the interrupt (via InterruptAttach() or InterruptAttachEvent()) and then blocks (via InterruptWait()), waiting for the ISR to tell it to do something (by returning an event of type SIGEV_INTR). To aid in minimizing scheduling latency, the interrupt service thread should raise its priority appropriately.

I/O message

   A message that relies on an existing binding between the client and the resource manager. For example, an _IO_READ message depends on the client’s having previously established an association (or context) with the resource manager by issuing an open() and getting back a file descriptor. See also connect message, context, combine message, and message.
I/O privileges

A particular right, that, if enabled for a given thread, allows the thread to perform I/O instructions (such as the x86 assembler `in` and `out` instructions). By default, I/O privileges are disabled, because a program with it enabled can wreak havoc on a system. To enable I/O privileges, the thread must be running as `root`, and call `ThreadCtl()`.

IPC

Interprocess Communication — the ability for two processes (or threads) to communicate. QNX Neutrino offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), as well as signals.

IPL

Initial Program Loader — the software component that either takes control at the processor’s reset vector (e.g. location 0xFFFFFFF0 on the x86), or is a BIOS extension. This component is responsible for setting up the machine into a usable state, such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also BIOS extension signature and startup code.

IRQ

Interrupt Request — a hardware request line asserted by a peripheral to indicate that it requires servicing by software. The IRQ is handled by the PIC, which then interrupts the processor, usually causing the processor to execute an Interrupt Service Routine (ISR).

ISR

Interrupt Service Routine — a routine responsible for servicing hardware (e.g. reading and/or writing some device ports), for updating some data structures shared between the ISR and the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.
kernel

See microkernel.

level-sensitive

One of two ways in which a PIC (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast edge-sensitive.

linearly mapped

A term indicating that a certain memory component is entirely addressable by the processor. Contrast bank-switched.

message

A parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message — the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various “changes of state” that affect when, and for how long, they may run.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mount structure

An optional, well-defined data structure (of type iofunc_mount_t) within an iofunc_*() structure, which contains information used on a
per-mountpoint basis (generally used only for filesystem resource managers). See also attributes structure and OCB.

**mountpoint**

The location in the pathname space where a resource manager has “registered” itself. For example, the serial port resource manager registers mountpoints for each serial device (/dev/ser1, /dev/ser2, etc.), and a CD-ROM filesystem may register a single mountpoint of /cdrom.

**multicore system**

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

**mutex**

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (pthread_mutex_lock()) and released (pthread_mutex_unlock()) around the code that accesses the shared data (usually a critical section). See also critical section.

**name resolution**

In a QNX Neutrino network, the process by which the Qnet network manager converts an FQNN to a list of destination addresses that the transport layer knows how to get to.

**name resolver**

Program code that attempts to convert an FQNN to a destination address.

**NDP**

Node Discovery Protocol — proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.
network directory

A directory in the pathname space that’s implemented by the Qnet network manager.

Neutrino

Name of an OS developed by QNX Software Systems.

NFS

Network File System — a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g. ls, cp, mv) operate on the remote files exactly as they do on your local files.

NMI

Nonmaskable Interrupt — an interrupt that can’t be masked by the processor. We don’t recommend using an NMI!

Node Discovery Protocol

See NDP.

node domain

A character string that the Qnet network manager tacks onto the nodename to form an FQNN.

nodename

A unique name consisting of a character string that identifies a node on a network.

nonbootable

A nonbootable OS image is usually provided for larger embedded systems or for small embedded systems where a separate, configuration-dependent setup may be required. Think of it as a second “filesystem” that has some additional files on it. Since it’s
nonbootable, it typically won’t contain the OS, startup file, etc. Contrast **bootable**.

**OCB**

Open Control Block (or Open Context Block) — a block of data established by a resource manager during its handling of the client’s `open()` function. This context block is bound by the resource manager to this particular request, and is then automatically passed to all subsequent I/O functions generated by the client on the file descriptor returned by the client’s `open()`.

**package filesystem**

A virtual filesystem manager that presents a customized view of a set of files and directories to a client. The “real” files are present on some medium; the package filesystem presents a virtual view of selected files to the client.

**partition**

A division of CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

**pathname prefix**

See **mountpoint**.

**pathname space mapping**

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

**persistent**

When applied to storage media, the ability for the medium to retain information across a power-cycle. For example, a hard disk is a persistent storage medium, whereas a ramdisk is not, because the data is lost when power is lost.
Photon microGUI

The proprietary graphical user interface built by QNX Software Systems.

PIC

Programmable Interrupt Controller — hardware component that handles IRQs. See also edge-sensitive, level-sensitive, and ISR.

PID

Process ID. Also often pid (e.g. as an argument in a function call).

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface — the “X” alludes to “UNIX”, on which the interface is based.

POSIX layer calls

Convenient set of library calls for writing resource managers. The POSIX layer calls can handle even more of the common-case messages and functions than the base layer calls. These calls are identified by the `iofunc_*()` prefix. In order to use these (and we strongly recommend that you do), you must also use the well-defined POSIX-layer attributes `iofunc_attr_t`, OCB `iofunc_ocb_t`, and (optionally) mount `iofunc_mount_t` structures.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been “preempted” by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY, the lower-priority thread is immediately preempted by the higher-priority thread.
prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent priority inversion.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU on behalf of the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it’s working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one thread running in it — this thread is then called the first thread.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.
process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of 1.

pty

Pseudo-TTY — a character-based device that has two “ends”: a master end and a slave end. Data written to the master end shows up on the slave end, and vice versa. These devices are typically used to interface between a program that expects a character device and another program that wishes to use that device (e.g. the shell and the telnet daemon process, used for logging in to a system over the Internet).

pulses

In addition to the synchronous Send/Receive/Reply services, QNX Neutrino also supports fixed-size, nonblocking messages known as pulses. These carry a small payload (four bytes of data plus a single byte code). A pulse is also one form of event that can be returned from an ISR or a timer. See MsgDeliverEvent() for more information.

Qnet

The native network manager in QNX Neutrino.

QoS

Quality of Service — a policy (e.g. loadbalance) used to connect nodes in a network in order to ensure highly dependable transmission. QoS is an issue that often arises in high-availability (HA) networks as well as realtime control systems.
RAM

Random Access Memory — a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast flash and EPROM.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data — you don’t want any translations of the raw binary stream between the device and the application. Contrast canonical mode.

replenishment

In sporadic scheduling, the period of time during which a thread is allowed to consume its execution budget.

reset vector

The address at which the processor begins executing instructions after the processor’s reset line has been activated. On the x86, for example, this is the address 0xFFFFFFF0.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX Neutrino resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g. serial ports, parallel ports, network cards, disk drives) or virtual (e.g. /dev/null, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with device drivers. But unlike device drivers, QNX Neutrino resource managers don’t require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also device driver.
RMA

Rate Monotonic Analysis — a set of methods used to specify, analyze, and predict the timing behavior of realtime systems.

round robin

Scheduling algorithm whereby a thread is given a certain period of time to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also adaptive, FIFO, and sporadic.

runmask

A bitmask that indicates which processors a thread can run on. Contrast inherit mask.

runtime loading

The process whereby a program decides while it’s actually running that it wishes to load a particular function from a library. Contrast static linking.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as “T_{sl}”. Contrast interrupt latency.

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session.
membership via `setsid()`. A session can contain multiple process groups.

**session leader**

A process whose death causes all processes within its process group to receive a SIGHUP signal.

**soft thread affinity**

The scheme whereby the microkernel tries to dispatch a thread to the processor where it last ran, in an attempt to reduce thread migration from one processor to another, which can affect cache performance. Contrast **hard thread affinity**.

**software interrupts**

Similar to a hardware interrupt (see **interrupt**), except that the source of the interrupt is software.

**sporadic**

Scheduling algorithm whereby a thread’s priority can oscillate dynamically between a “foreground” or normal priority and a “background” or low priority. A thread is given an execution **budget** of time to be consumed within a certain **replenishment** period. See also **adaptive**, **FIFO**, and **round robin**.

**startup code**

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

**static bootfile**

An image created at one time and then transmitted whenever a node boots. Contrast **dynamic bootfile**.
static linking

The process whereby you combine your modules with the modules from the library to form a single executable that’s entirely self-contained. The word “static” implies that it’s not going to change — all the required modules are already combined into one.

symmetric multiprocessing (SMP)

A multiprocessor system where a single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under QNX Neutrino. A thread is a flow of execution; it exists within the context of a process.

timer

A kernel object used in conjunction with time-based functions. A timer is created via `timer_create()` and armed via `timer_settime()`. A timer can then deliver an event, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a round-robin or adaptive scheduled thread. This period of time is small (on the order of tens of milliseconds); the actual value shouldn’t be relied upon by any program (it’s considered bad design).
Index

! (MQ_PRIO_MAX-1) 1751
.rhosts 2657
/dev/name/global 1880
/dev/name/local 1880
/dev/zero 1680
/etc/autoconnect 1910
/etc/hosts 521, 864, 866, 868–871, 875, 876, 880, 1029
/etc/hosts.equiv 2657
/etc/networks 900, 902, 904, 1917, 2884
/etc/protocols 547, 926, 928, 930, 2123, 2884
/etc/resolv.conf 2517, 2519, 2522, 2525, 2528, 2531
/etc/services 550, 956, 958, 960, 2839, 2909
_BEGIN_DECLS 113
_BIGENDIAN 113
_cabsargs, _cabsfargs 257
_CHAR_SIGNED 113
_CHAR_UNSIGNED 113
_DIOF() 424
_DION() 424
_DIOI() 424
_DIOF() 424
_END_DECLS 113
_FILE__ 168
_INT_BITS__ 113
_LINE__ 168
_LITTLEENDIAN__ 113
_LONG_BITS__ 113
_OPTIMIZE__ 113
_progname 2122
_PTR_BITS__ 113
_QNX__ 113
_QNXNTO__ 113
_res_state 2515
_signalstub() 2987
_ambksiz 157, 1589, 2662, 3068
_argc 159
_argv 160
_asyncmsg_connection_attr 183
_auxv 233
_btext 252
_client_info 385
_clockadjust 346
_clockperiod 354
Index

© 2007, QNX Software Systems GmbH & Co. KG.

疔cmdfd() 367
𝙨_Cmdname() 368
cred_info 385
CS_ARCHITECTURE 371
CS_DOMAIN 372
CS_HOSTNAME 881, 2865
CS_HOSTNAME 372
CS_HW_PROVIDER 372
CS_HW_SERIAL 372
CS_LIBPATH 372
CS_MACHINE 372
CS_PATH 372
CS_RELEASE 372
CS_RESOLVE 372
CS_SRPC_DOMAIN 372
CS_SYSNAME 372
CS_TIMEZONE 3581
CS_TIMEZONE 372
CS_VERSION 372
DCMD_ALL 425
DCMD_BLK 425
DCMD_CAM 425
DCMD_CHR 425
DCMD_FSYS 425
DCMD_INPUT 425
DCMD_IP 425
DCMD_MEM 425
DCMD_MISC 425
DCMD_MIXER 425
DCMD_NET 425
DCMD_PHOTON 425
DCMD_PROC 425
DEVCTL_DATA() 1255
Déxtra_first() 2465
Déxtra_next() 2465
Déxtra_valid() 2465
DTYPE_NONE 2465
DTYPE_STAT 2465
edata 516
end 519
etext 576
exit() 624
fdinfo 1259, 1263
_fDINFO_FLAG_LOCALPATH
1212, 1262
FP_EXC_DENORMAL 725, 728
FP_EXC_DIVZERO 725, 728
FP_EXC_INEXACT 725, 728
FP_EXC_INVALID 725, 728
FP_EXC_OVERFLOW 725, 728
FP_EXC_UNDERFLOW 725, 728
FP_PREC_DOUBLE 731
FP_PREC_DOUBLE_EXTENDED
731
FP_PREC_EXTENDED 731
FP_PREC_FLOAT 731
FP_ROUND_LEAREST 734
FP_ROUND_NEGATIVE 734
FP_ROUND_POSITIVE 734
FP_ROUND_ZERO 734
FTYPE_ANY 1199, 1205, 1207,
2533
FTYPE_LINK 1199, 1205, 1207,
2533
FTYPE_MOUNT 1199, 1205,
1207, 2533
FTYPE_QUEUE 1199, 1205,
1207, 2533
FTYPE_PIPE 1199, 1205, 1207,
2533
FTYPE_NORMAL 1199, 1205, 1207,
2533
FTYPE_UNDERFLOW 1199, 1205,
1207, 2533
FTYPE_OVERFLOW 1199, 1205,
1207, 2533
FTYPE_INVALID 1199, 1205,
1207, 2533
FTYPE_INEXACT 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NORM 1199, 1205,
1207, 2533
FTYPE_ZERO 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
1207, 2533
FTYPE_NAN 1199, 1205,
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>_FTYPE_SHMEM</td>
<td>1199, 1205,</td>
</tr>
<tr>
<td></td>
<td>1208, 2534</td>
</tr>
<tr>
<td>_FTYPE_SOCKET</td>
<td>1199, 1205,</td>
</tr>
<tr>
<td></td>
<td>1208, 2534</td>
</tr>
<tr>
<td>_FTYPE_SYMLINK</td>
<td>1199, 1205,</td>
</tr>
<tr>
<td></td>
<td>1208, 2534</td>
</tr>
<tr>
<td>_input_line_max</td>
<td>1152</td>
</tr>
<tr>
<td>intr_v86()</td>
<td>1194</td>
</tr>
<tr>
<td>_intrspin_t</td>
<td>1169, 1182,</td>
</tr>
<tr>
<td></td>
<td>1187</td>
</tr>
<tr>
<td>IO_CHMOD</td>
<td>1232, 1235</td>
</tr>
<tr>
<td>IO_CHOWN</td>
<td>1237, 1240</td>
</tr>
<tr>
<td>IO_CLOSE</td>
<td>1244, 1247</td>
</tr>
<tr>
<td>IO_COMBINE_FLAG</td>
<td>1233,</td>
</tr>
<tr>
<td></td>
<td>1238, 1245,</td>
</tr>
<tr>
<td></td>
<td>1255, 1262,</td>
</tr>
<tr>
<td></td>
<td>1284, 1292,</td>
</tr>
<tr>
<td></td>
<td>1300, 1331,</td>
</tr>
<tr>
<td></td>
<td>1337, 1345,</td>
</tr>
<tr>
<td></td>
<td>1355, 1361,</td>
</tr>
<tr>
<td></td>
<td>1368, 1383,</td>
</tr>
<tr>
<td></td>
<td>1392</td>
</tr>
<tr>
<td>io_connect</td>
<td>1198</td>
</tr>
<tr>
<td>IO_CONNECT</td>
<td>1327</td>
</tr>
<tr>
<td>IO_CONNECT_COMBINE</td>
<td>1198</td>
</tr>
<tr>
<td>IO_CONNECT_COMBINE_CLOSE</td>
<td>1198</td>
</tr>
<tr>
<td>IO_CONNECT_EFLAG_DIR</td>
<td>1203, 1208</td>
</tr>
<tr>
<td>IO_CONNECT_EFLAG_DOT</td>
<td>1203, 1208</td>
</tr>
<tr>
<td>IO_CONNECT_EXTRA_LINK</td>
<td>1203</td>
</tr>
<tr>
<td>IO_CONNECT_EXTRA_MOUNT</td>
<td>1204</td>
</tr>
<tr>
<td>IO_CONNECT_EXTRA_MOUNT_OCB</td>
<td>1204</td>
</tr>
<tr>
<td>IO_CONNECT_EXTRA_MQUEUE</td>
<td>1203</td>
</tr>
<tr>
<td>IO_CONNECT_EXTRA_NONE</td>
<td>1203</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_PHOTON</td>
<td>1204</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_PROC_SYMLINK</td>
<td>1204</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_RENAME</td>
<td>1204</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_RESMGR_LINK</td>
<td>1204</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_SEM</td>
<td>1204</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_SOCKET</td>
<td>1204</td>
</tr>
<tr>
<td>_IO_CONNECT_EXTRA_SYMLINK</td>
<td>1203</td>
</tr>
<tr>
<td>_io_connect_ftype_reply</td>
<td>1205</td>
</tr>
<tr>
<td>_IO_CONNECT_LINK</td>
<td>1199</td>
</tr>
<tr>
<td>_io_connect_link_reply</td>
<td>1207</td>
</tr>
<tr>
<td>_IO_CONNECT_MKNOD</td>
<td>1199</td>
</tr>
<tr>
<td>_IO_CONNECT_MOUNT</td>
<td>1199</td>
</tr>
<tr>
<td>_IO_CONNECT_OPEN</td>
<td>1198</td>
</tr>
<tr>
<td>_IO_CONNECT_READLINK</td>
<td>1199</td>
</tr>
<tr>
<td>_IO_CONNECT_RENAME</td>
<td>1199, 1351</td>
</tr>
<tr>
<td>_IO_CONNECT_RET_LINK</td>
<td>1323</td>
</tr>
<tr>
<td>_IO_CONNECT_RSVD_UNBLOCK</td>
<td>1199</td>
</tr>
<tr>
<td>_IO_CONNECT_UNLINK</td>
<td>1198</td>
</tr>
<tr>
<td>_IO_DEVCTL</td>
<td>1253, 1257</td>
</tr>
<tr>
<td>_IO_FDINFO</td>
<td>1260, 1261</td>
</tr>
<tr>
<td>_IO_FLAG_RD</td>
<td>1363</td>
</tr>
<tr>
<td>_IO_FLAG_WR</td>
<td>1363</td>
</tr>
<tr>
<td>_IO_LOCK</td>
<td>1276</td>
</tr>
<tr>
<td>_IO_LSEEK</td>
<td>1283, 1286</td>
</tr>
<tr>
<td>_IO_MMAP</td>
<td>1291, 1295</td>
</tr>
<tr>
<td>_IO_OPENFD</td>
<td>1330, 1334</td>
</tr>
</tbody>
</table>
Index

© 2007, QNX Software Systems GmbH & Co. KG.

_IO_OPENFD_NONE 1332
_IO_OPENFD_PIPE 1332
_IO_OPENFD_RESERVED 1332
_IO_PATHCONF 1336, 1339
_IO_READ 1341, 1344
_IOC_SET_CONNECT_RET() 1323
_IOC_SET_PATHCONF_VALUE() 1336
_IOC_SET_READ_NBYTES() 1344
_IOC_SET_WRITE_NBYTES() 1391
_IOC_SPACE 1355
_IOC_STAT 1358, 1360
_IOC_SYNC 1365, 1367
_IOC_UTIME 1383, 1385
_IOC_WRITE 1388
_IOC_XFLAG_BLOCK 1345, 1392
_IOC_XFLAG_DIR_EXTRA_HINT 1345, 1392
_IOC_XFLAG_NONBLOCK 1345, 1392
_IOC_XTYPE_MQUEUE 1345, 1392
_IOC_XTYPE_NONE 1345, 1392
_IOC_XTYPE_OFFSET 1345, 1392
_IOC_XTYPE_READCOND 1345, 1392
_IOC_XTYPE_REGISTRY 1345, 1392
_IOC_XTYPE_TCPPIP 1345, 1392
_IOC_XTYPE_TCPPIP_MSG 1345, 1392
_IOFBF 2926
_IOLBF 2926
_IONBF 2926
__timer 3502
_MAJOR_BLK_PREFIX 2649
_MAJOR_CHAR_PREFIX 2649
_MAJOR_DEV 2649
_MAJOR_FSYS 2649
_MAJOR_PATHMGR 2649
_MFLAG_OCB 1713, 1716
_MOUNT_AFTER 1713, 1716
_MOUNT_ATIME 1716
_MOUNT_BEFORE 1713, 1716
_MOUNT_CREAT 1716
_MOUNT_ENUMERATE 1714, 1716
_MOUNT_FORCE 1714, 1716, 3595
_MOUNT_NOATIME 1713, 1716
_MOUNT_NOCREAT 1713, 1716
_MOUNT_NOEXEC 1254, 1713, 1716
_MOUNT_NOSUID 1254, 1713, 1716
_MOUNT_OFF32 1713, 1716
_MOUNT_OPAQUE 1714, 1716
_MOUNT_READONLY 1254, 1713, 1716
_MOUNT_REMOUNT 1714, 1716
_MOUNT_SUID 1716
_MOUNT_UNMOUNT 1714, 1716
_MSG_INFO 1759, 1773
_NOTIFY_ACTION_POLL 1396, 1397
_NOTIFY_ACTION_POLLARM 1397
_NOTIFY_ACTION_TRANARM 1397
_NOTIFY_COND_INPUT 1300, 1395
_NOTIFY_COND_MASK 1395
_NOTIFY_COND_OBAND 1301, 1395
_NOTIFY_COND_OUTPUT 1301, 1395
_NOTIFY_DATA_MASK 1396
_NTO_CHF_COID_DISCONNECT 302
_NTO_CHF_DISCONNECT 303
_NTO_CHF_FIXED_PRIORITY 301, 303
_NTO_CHF_NET_MSG 303
_NTO_CHF_REPLY_LEN 303, 1760
_NTO_CHF_SENDER_LEN 304, 1760
_NTO_CHF_THREAD_DEATH 304
_NTO_CHF_UNBLOCK 304, 305, 1760
_NTO_COF_CLOEXEC 380, 390
_NTO_INTR_CLASS_EXTERNAL 1155
_NTO_INTR_CLASS_SYNTHETIC 1155
_NTO_INTR_FLAGS_END 1159, 1160, 1165, 1177
_NTO_INTR_FLAGS_PROCESS 1159, 1160, 1165, 1177, 1178
_NTO_INTR_FLAGS_TRK_MSK 1159, 1177
_NTO_INTR_SPARSE 1155
_NTO_KEYDATA_CALCULATE 1775, 1778
_NTO_KEYDATA_VERIFY 1775, 1778
_NTO_MI_UNBLOCK_REQ 1760
_NTO_RESET_OVERRUNS 3497
_NTO_SCTL_GETPrioCeiling 3309
_NTO_SCTL_SETEVENT 3309
_NTO_SCTL_SETPrioCeiling 3309
_NTO_SIDE_CHANNEL 380
_NTO_SYNC_COND 3330
_NTO_SYNC_MUTEX_FREE 3330
_NTO_SYNC_SEM 3330
_NTO_TCTL_ALIGN_FAULT 3447
_NTO_TCTL_IO 1155, 1164, 1169, 1172, 1174, 1182, 1184, 1187, 1189, 3449
_NTO_TCTL_RUNMASK 3450
_NTO_TCTL_THREADS_CONT 3451
_NTO_TCTL_THREADS_HOLD 3451
_NTO_TIMEOUT_CONDVAR 3507
_NTO_TIMEOUT_INTR 3507
_NTO_TIMEOUT_JOIN 3507
_NTO_TIMEOUT_MUTEX 3507
_NTO_TIMEOUT_RECEIVE 3507
_NTO_TIMEOUT_REPLY 3507
_NTO_TIMEOUT_SEM 3507
_NTO_TIMEOUT_SEND 3508
_NTO_TIMEOUT_Sigsuspend 3508
_NTO_TIMEOUT_SIGWAITINFO 3508
_NTO_TIMER_SEARCH 3497
_NTO_TRACE * 3540
<table>
<thead>
<tr>
<th>Index</th>
<th>© 2007, QNX Software Systems GmbH &amp; Co. KG.</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NTO_VERSION</em> 113</td>
<td><em>POSIX THREAD_SAFE_FUNCTIONS</em> 851, 893</td>
</tr>
<tr>
<td><em>PATH_UTMP</em> 553</td>
<td><em>pulse</em> 2377</td>
</tr>
<tr>
<td><em>PC_CHOWN_RESTRICTED</em> 737, 1981</td>
<td><em>PULSE_CODE_COIDDEATH</em> 302, 2377</td>
</tr>
<tr>
<td><em>PC_LINK_MAX</em> 736, 1980</td>
<td><em>PULSE_CODE_DISCONNECT</em> 303, 2377</td>
</tr>
<tr>
<td><em>PC_MAX_CANON</em> 736, 1980</td>
<td><em>PULSE_CODE_MAXAVAIL</em> 1825, 2377, 2970</td>
</tr>
<tr>
<td><em>PC_MAX_INPUT</em> 736, 1980</td>
<td><em>PULSE_CODE_MINAVAIL</em> 1825, 2377, 2970</td>
</tr>
<tr>
<td><em>PC_NAME_MAX</em> 736, 1980</td>
<td><em>PULSE_CODE_NET_ACK</em> 2377</td>
</tr>
<tr>
<td><em>PC_NO_TRUNC</em> 737, 1981</td>
<td><em>PULSE_CODE_NET_DETACH</em> 2377</td>
</tr>
<tr>
<td><em>PC_PATH_MAX</em> 737, 1981</td>
<td><em>PULSE_CODE_NET_UNBLOCK</em> 2377</td>
</tr>
<tr>
<td><em>PC_PIPE_BUF</em> 737, 1981</td>
<td><em>PULSE_CODE_THREADDEATH</em> 304, 457, 2377</td>
</tr>
<tr>
<td><em>PC_VDISABLE</em> 737, 1981, 3416</td>
<td><em>PULSE_CODE_UNBLOCK</em> 304, 457, 2377</td>
</tr>
<tr>
<td><em>PCCARD_ARM_INSERT_REMOVE</em> 1993</td>
<td><em>PULSE_SUBTYPE</em> 2377</td>
</tr>
<tr>
<td><em>PCCARD_DEV_AIMS</em> 1993</td>
<td><em>PULSE_TYPE</em> 2377</td>
</tr>
<tr>
<td><em>PCCARD_DEV_ALL</em> 1993</td>
<td><em>RESMGIR_CONNECT_NFUNCS</em> 1265</td>
</tr>
<tr>
<td><em>PCCARD_DEV_FIXED_DISK</em> 1993</td>
<td><em>RESMGIR_DEFAULT</em> 1372</td>
</tr>
<tr>
<td><em>PCCARD_DEV_GPIB</em> 1993</td>
<td><em>RESMGIR_DETACH_ALL</em> 2554</td>
</tr>
<tr>
<td><em>PCCARD_DEV_MEMORY</em> 1993</td>
<td><em>RESMGIR_DETACH_PATHNAME</em> 2554</td>
</tr>
<tr>
<td><em>PCCARD_DEV_NETWORK</em> 1993</td>
<td><em>RESMGIR_FLAG_AFTER</em> 2537</td>
</tr>
<tr>
<td><em>PCCARD_DEV_PARALLEL</em> 1993</td>
<td><em>RESMGIR_FLAG_BEFORE</em> 2537</td>
</tr>
<tr>
<td><em>PCCARD_DEV_SCSI</em> 1993</td>
<td><em>RESMGIR_FLAG_DIR</em> 2537</td>
</tr>
<tr>
<td><em>PCCARD_DEV_SERIAL</em> 1993</td>
<td><em>RESMGIR_FLAG_FTYPEONLY</em> 2538</td>
</tr>
<tr>
<td><em>PCCARD_DEV_SOUN D</em> 1993</td>
<td><em>RESMGIR_FLAG_OPAQUE</em> 2538</td>
</tr>
<tr>
<td><em>PCCARD_DEV_VIDEO</em> 1993</td>
<td><em>RESMGIR_FLAG_SELF</em> 2539</td>
</tr>
<tr>
<td><em>pcocard_info</em> 2001</td>
<td><em>resmgr_handle_grow()</em> 2561</td>
</tr>
<tr>
<td><em>PCCARD_MEMTYPE_ATTRIBUTE</em> 2006</td>
<td></td>
</tr>
<tr>
<td><em>PCCARD_MEMTYPE_COMMON</em> 2006</td>
<td></td>
</tr>
<tr>
<td><em>POSIX_AIO_MAX</em> 1527</td>
<td></td>
</tr>
<tr>
<td><em>POSIX_CHOWN_RESTRICTED</em> 317, 645, 1509</td>
<td></td>
</tr>
<tr>
<td><em>POSIX_LOGIN_NAME_MAX</em> 893</td>
<td></td>
</tr>
</tbody>
</table>
_resmgr_io_func() 2569
_RESMGR_IO_NFUNCTIONS 1265
_RESMGR_NOREPLY 1372
_RESMGR_NPARTS() 2586
_resmgr_ocb() 2588
_RESMGR_PATHNAME_LOCALPATH 2593
_RESMGR_PTR() 2596
_RESMGR_STATUS() 2598
_SC_ARG_MAX 3334
_SC_CHILD_MAX 3334
_SC_CLK_TCK 3334
_SC_GETGR_R_SIZE_MAX 851
_SC_GETPW_R_SIZE_MAX 937, 942, 977, 981
_SC_JOB_CONTROL 3335
_SC_NGROUPS_MAX 3334
_SC_OPEN_MAX 3335
_SC_PAGESIZE 2155
_SC_SAVED_IDS 3335
_SC_VERSION 3335
_scalloc() 2671
_sched_info 2751
_server_info 394
_sfree() 2929
_SIGMAX 2990
_SIGMIN 2990
_sleepon_broadcast() 3046
_sleepon_destroy() 3048
_sleepon_init() 3050
_sleepon_lock() 3052
_sleepon_signal() 3054
_sleepon_unlock() 3056
_sleepon_wait() 3058
_SLOG_CRITICAL 3063
_SLOG_DEBUG 3063
_SLOG_DEBUG1 3063
_SLOG_DEBUG2 3063
_SLOG_ERROR 3063
_SLOG_INFO 3063
_SLOG_NOTICE 3063
_SLOG_SETCODE() 3062, 3672
_SLOG_SHUTDOWN 3063
_SLOG_WARNING 3063
_smalloc() 3068
_srealloc() 3186
_syspage_ptr 3355
_thread_attr 3441
_thread_local_storage 3444
_timer_info 3498
_TLS 3444
8-bit characters, reading 1696
8086 mode, virtual 1194

A

abort() 119
abs() 121
absolute values
  complex number 257
  floating point 638
  integer 121
  long integer 1506
accept() 123, 3390, 3813
access() 126
ACCOUNTING 3633
acos(), acosf() 129
acosh(), acoshf() 131
ACTION 1031
adaptive partitions
  averaging window size, setting 355, 2715
  bankruptcy, handling 2717
budgets
  zero, preventing 2741, 2743
  zero, setting on
    bankruptcy 2718
CPU time, getting 2730
creating 2721
IDs, finding for a
  partition 2725
inheritance, preventing 303
locking 2740
maximum number of 2723
names
  assigning 2721
  maximum length 2721
parameters, setting 2728
processes
  getting 2744
  setting 2726
querying 2723
scheduler
  controlling 2713
  parameters, getting 2714
  parameters, setting 2720
  scheduling policies 2716
security, setting 2740
System 2721
threads
  critical 2734, 2735, 2972
  determining 2736
  joining 2726, 2743
addresses
hosts
  strings, converting
to/from 1106, 1109
IP
  strings, converting
to/from 1085, 1087, 1102,
1104, 1106, 1109
IPv6 1114
link-local 1115
local network
  IP addresses, converting
to/from 1091
  IP addresses, extracting
  from 1089
  nodenames, translating 895
  scoped 1115
  site-local 1115
  sockets 133, 756, 811, 816
addrinfo 133
  errors 811
  freeing 756
  getting 816
advisory locks  See files, locking
AF_INET 864, 867, 870, 900, 1029,
1054, 1106, 1109, 1400,
1917, 3390, 3587
AF_INET6 870, 1056, 1106, 1109,
1423
AF_LOCAL 3616
AF_UNSPEC 2621
AH (Authentication Header) 1406
AIMS (Auto Incrementing Mass
  Storage) 1993
aio_cancel() 135
aio_error() 137
aio_fsync() 139
aio_read() 141
aio_return() 143
aio_suspend() 145
aio_write() 147
aiocb 1528
alarm() 149
alarms, scheduling 149, 3584
aligned memory, allocating 1621, 2083
alignment, setting for a process 3122
alloca() 152
alphabetic, testing a character for 1435, 1466
alphanumeric, testing a character for 1433, 1464
alphasort() 155
ANSI classification 105
APS_INIT_DATA() 2714
APS_PARTITION_NAME_LENGTH 2721
APS_QCRIT_BILL_AS_CRITICAL 2737
APS_QCRIT_PERM_CRITICAL 2736
APS_QCRIT_RUNNING_CRITICAL 2737
APS_SCHED_SEC_OFF 2740
APS_SYSTEM_PARTITION_NAME 2721
arccosines 129
architecture, instruction set 371
arcsines 164
arctangents 196, 198
argument lists, variable-length 3639, 3645, 3647, 3649
coercion 3640
arguments to main() 159, 160, 233
parsing 906
arrays allocating 273, 2671
quick-sorting 2417
searching 249, 1517, 1563
ASCII, testing a character for 1437
asctime(), asctime_r() 161
asin(), asinf() 164
asinh(), asinhf() 166
assert() 168
asynchronous connection attributes
_asyncmsg_connection_attr 183
message
_asyncmsg_put(), asyncmsg_putv() 194
messages
_asyncmsg_channel_create() 172
_asyncmsg_channel_destroy() 174
asynchronous I/O
canceling 135
error status, getting 137
file, synchronizing 139
reading 141
return status, getting 143
waiting for completion 145
writing 147
asynchronous messages
buffers freeing 187
channels 179
attaching to a process 177
creating 172
destroying 174
flushing 185
receiving 189

## Index

© 2007, QNX Software Systems GmbH & Co. KG.

asynchronous SNMP
- transactions 3085
  - asyncmsg_channel_create() 172
  - asyncmsg_channel_destroy() 174
  - asyncmsg_connect_attach() 177
  - asyncmsg_connect_attr() 179
  - asyncmsg_connect_detach() 181
  - asyncmsg_flush() 185
  - asyncmsg_free() 187
  - asyncmsg_get() 189
  - asyncmsg_malloc() 191
  - asyncmsg_put(), asyncmsg_putv() 194
  - atan(), atanf() 196
  - atan2(), atan2f() 198
  - atanh(), atanhf() 200
  - atexit() 202
  - atof() 205
  - atoi() 207
  - atol(), atoll() 211
  - atomic_add_value() 215
  - atomic_add() 213
  - atomic_clr_value() 219
  - atomic_clr() 217
  - atomic_set_value() 223
  - atomic_set() 221
  - atomic_sub_value() 227
  - atomic_sub() 225
  - atomic_toggle_value() 231
  - atomic_toggle() 229

atomic operations
- addition 213, 215
- bits
  - clearing 217, 219
  - setting 221, 223
  - toggling 229, 231

subtraction 225, 227

attributes
- connection 183
  - Authentication Header (AH) 1406
  - authenticator() 3091
  - Auto Incrementing Mass Storage (AIMS) 1993

B

background processes 413, 2106
- termination, notification of 2108

bankruptcies
- handling 2717

BARRIER_SERIAL_THREAD 2177

barriers
- attributes 2175
  - destroying 2179
  - initializing 2183
  - process-shared 2181, 2185
  - destroying 2173
  - initializing 2175

basename() 234

bcmp() 237

bcopy() 239

Bessel functions
- first kind 1493, 1495, 1497
- second kind 3802, 3804, 3806

big endian
- __BIGENDIAN__ manifest 113
little endian, converting
to/from 534, 536, 538,
540, 542, 544, 3291
messages 1640
native format, converting
to/from 522, 524, 526
ports
reading from 1075, 1079
writing to 1969, 1973
unaligned values
accessing safely 3603, 3605,
3607
writing safely 3597, 3599,
3601
binary search 249
bind() 241, 3390, 3813
bindresvport() 244
BIOS (PCI), determining if
present 2038
bits
atomic operations
clearing 217, 219
setting 221, 223
toggling 229, 231
set, finding first 672
block buffering, setting for stream
I/O 2842
block special devices
reading 2455
writing 3794
blocks
allocating 3068
reading from a file 2455
system message log, writing
to 3060
writing to a file 3794
blocksize, filesystem 777, 3199
BOOT_TIME 995, 3632
booting 3348
time since 3352
break condition, asserting 3393
break pointer
advancing 2662
increment 157
brk() 246
BRKINT 3413
bsearch() 249
BSS data 516, 519
btowc() 253
budgets
setting 2728
zero
bankruptcy, setting on 2718
preventing 2741, 2743
buffer
controlling 3540
buffers
canonical input 736, 1980
freeing 187
locking 1867
raw input 736, 1980
stream I/O 2840, 2926
block 2842
flushing 3374
line 2877
BUFSIZ 2840
BULK_REQ_MSG 3080
bus mastering, enabling 2017
busy-waiting 1896, 1898, 1901,
1903, 1905
bytes
comparing 237
copying 239, 1623, 1629
overlapping objects 1635
multibyte character, number of bytes in 1597, 1600
reading 2094, 2446
reordering 1037, 1039, 1940, 1942
setting 1637
writing 2411, 3789
zeroing 255

bzero() 255

C

C_ANY 2518, 2521, 2524, 2527
C_CHAOS 2518, 2521, 2524, 2527
C_HS 2518, 2521, 2524, 2527
C_IN 2518, 2521, 2524, 2527

C++ programs
end of C code 113
start of C code 113
cabs(), cabsf() 257

calendar times
current 3463
local times, converting
to/from 1539, 1541, 1668
tm 3520
callback() 3092
calloc() 273
cancellation
cleanup handlers 2189, 2191
points 3444
creating 2373
state 2333, 3437
type 2335, 3437
canonical input buffer 736, 1980

Card Information Structure (CIS),
reading 2007
CardBus cards 2050
cbrt(), cbrtf() 275
ceil(), ceilf() 277
cfgetispeed() 279
cfgetospeed() 282
CFGFILE_APPEND 286
CFGFILE_CREAT 286
CFGFILE_EXCL 287
CFGFILE_RDONLY 286
CFGFILE_RDWR 286
CFGFILE_TRUNC 287
CFGFILE_WRONLY 286
cfgopen() 286
cfmakeraw() 290
cfree() 292
cfsetispeed() 294
cfsetospeed() 297

ChannelCreate(),
ChannelCreate_r() 300

ChannelDestroy(),
ChannelDestroy_r() 307

channels
creating 172, 300, 1880
destroying 174, 307, 1889
events, delivering 1762
flags 302
ID 301
messages
receiving 1791, 1805
replying 1809, 1812
sending 1816, 1821, 1830, 1834, 1838, 1842, 1846, 1850
pulses
receiving 1797, 1801
sending 1826
side channels 380
CHAR_MAX 1536
character device terminal drivers, providing session support to 2120
characters See also strings; wide characters
8-bit, reading 1696
control disabling 737, 1981
discarding on input 1696
default (signed or unsigned) 113
devices
input stream, injecting 3385
size 3383, 3404
escape 1708
handling 2880
international See wide characters
lowercase, converting to 1696, 3528
multibyte
bytes, counting 1597, 1600
wide characters, conversion object 1605
wide characters, converting to/from 1602, 1607, 1609, 1612, 3713, 3741, 3757, 3767
number waiting to be read 3388
searching for 1625, 3213, 3257
sets, searching for 3223, 3255, 3267
special 1708
stdin, reading from 675, 827, 829
stdout, writing to 743, 2390, 2392
streams
pushing back 3612
reading from 673, 679, 823, 825
writing to 741, 2386, 2388
testing for
alphabetic 1435
alphanumeric 1433
ASCII 1437
control character 1441
decimal digit 1443
hexadecimal digit 1488
lowercase 1451
printable 1447, 1455
punctuation 1457
uppercase 1462
whitespace 1459
uppercase, converting to 3530, 3536
wide characters, converting to/from 253, 3765
chdir() 309
chmod() 312, 1222
resource managers, implementing in 1232, 1235
chown() 316, 1222
resource managers, implementing in 1237, 1240
restricting use of 737, 1981
chroot() 319
chsize() 322
CIDR (Classless Internet Domain Routing) 1093, 1098
CIS (Card Information Structure), reading 2007
classes
  IP addresses 1094
  PCI 2026
  wide-character 3772
Classless Internet Domain Routing (CIDR) 1093, 1098
clearenv() 325
clearerr() 328
el 1172
CLK_TCK 3514
CLOCAL 3414
clock
  adjusting 347
  CPU time, getting ID of for a thread 2235
  cycles 349
  getting 358
  ID, getting 332
  period, getting and setting 355
  resolution, getting 334
  setting 343, 358
  ticks 3334
  per seconds 3352
  time, getting 336
clock_getcpu() 332
clock_getres() 334
clock_gettime() 336
CLOCK_MONOTONIC 339, 3480, 3488, 3491, 3505
clock_nanosleep() 340
CLOCK_REALTIME 339, 343, 346, 354, 358, 2266, 2310,
  2313, 2821, 3465, 3480, 3488, 3491, 3505
clock_settime() 343
CLOCK_SOFTTIME 339, 3465, 3480, 3488, 3491, 3505
clock_t 3334, 3513
clock() 330
ClockAdjust(), ClockAdjust_r() 347
ClockCycles() 349
ClockId(), ClockId_r() 351
ClockPeriod(), ClockPeriod_r() 355
CLOCKS_PER_SEC 330
ClockTime(), ClockTime_r() 358
close() 361
  resource managers, implementing in 1244, 1247
closedir() 363, 1954
closelog() 366
cmsghdr 3617
code, portability 105
collating sequence, setting 2880
COLUMNS 325
commands
  executing 3356
  on a remote host 2435, 2627
  options, parsing 906, 984
communications line
  break condition, asserting 3393
  disconnecting 3368
comparison
  bytes 237, 1627, 1633
  strings
Index

case-insensitive 1633, 3208, 3217, 3235
case-sensitive 1627, 3215
locale's collating sequence, using 3219
wide-character 3719, 3721, 3733, 3776
substrings
case-insensitive 3241, 3251
case-sensitive 3246
compiling with optimization 113
complementary error function 563
complex numbers, absolute value of 257
computer, rebooting 3348
condition variables
attributes
clock 2212, 2218
destroying 2210
initializing 2216
process-shared 2214, 2220
blocking on 2202, 2206
destroying 2196, 3312
initializing 2198, 3331
synchronization objects 3301, 3304
unblocking
all threads 2194
highest priority thread 2200
waiting on 2206
timed 2202
configuration files, opening 286, 641
configuration strings
_CS_HOSTNAME 881, 2865
_CS_TIMEZONE 3581
getting and setting 371
configuration values, getting 736, 1336, 1980, 3334
confstr() 371
connect functions (resource managers)
default values, setting 1265
open, default 1327
connect() 376, 3390, 3812
ConnectAttach(),
   ConnectAttach_r() 380
ConnectClientInfo(),
   ConnectClientInfo_r() 384
ConnectDetach(),
   ConnectDetach_r() 303, 388
ConnectFlags(), ConnectFlags_r() 390
connection
attributes
structure 183
buffers, attributes 179
connections
allocating 191
client, information about 384
detaching 181, 303, 388
dispatch interface,
   creating 1646
flags, modifying 390
ID 381
server, information about 393
shutting down
   full-duplex 2952
sockets
   accepting on 123, 2421
   initiating on 376, 1910, 2438
listening on 1532, 2615

ConnectServerInfo(),

  ConnectServerInfo_r() 393

console I/O

  stderr 69, 3203
  stdin 69, 3204
  stdout 69, 3205

const 104

contention scope 2140, 2163

control characters 3415

  disabling 737, 1981
  discarding on input 1696
  testing a character for 1441, 1468

controlling terminals

  making 3402
  path name 408

conventions

  typographical liii

copysign(), copysignf() 396

core files, maximum size 2903

cos(), cosf() 398

cosh(), coshf() 400

cosines 398

  hyperbolic 400
  inverse hyperbolic 131

CPU usage

  overall, getting 2732
  partition, getting for 2730

CREAD 3414

creat(), creat64() 402, 3593

critical threads

  determining if 2736
  marking as 2734, 2973

crypt() 406

CS5, . . . , CS8 3414

CSIZE 3414

CSTOPB 3414

ctermid() 408

ctime(), ctime_r() 410

CTL_MAXNAME 3343

CTL_NET 3338, 3339

cube roots 275

current working directory 309, 831, 1007

d

D

D_FLAG_FILTER 435

D_FLAG_STAT 436

daemon() 413

daemons

  SNMP (Simple Network Management Protocol),
  configuration file for 2451

  system 413, 2106
  termination, notification of 2108

data segment

  changing space allocated for 246
  end of 516, 519
  maximum size 2903

data server

  applications
    deregistering 497
    registering 503

  variables
    creating 495
    deleting 492
flags, setting 499
getting 501
setting 505
data streams, flow control 3371
databases
blocks
reading 2455
writing 3794
groups
closing 520
ID, getting information about 849, 851
membership 1146
name, getting information about 854, 856
next entry, getting 846
rewinding 2859
hosts
closing 521
entries, getting 864, 868, 870, 874, 876, 879
errors 1020, 1026, 1035
opening 2863
structure 1029
network
closing 546
entries, getting 900, 902, 904
opening 2884
structure 1917
passwords
closing 548
encrypting 406, 2414
entry, getting for a user 935, 938, 940, 943
entry, getting next 932
rewinding 2895
protocols
closing 547
entries, getting 926, 928, 930
opening 2893
protoent 2123
services
closing 550
entries, getting 956, 958, 960
entry structure 2839
opening 2909
shadow passwords
closing 551
entry, reading 682, 977, 981
entry, structure 2399
entry, writing 2399
rewinding 2916
system packet forwarding See ROUTE
datagrams 3101, 3102, 3587
date 3581
daylight 415, 3581
daylight saving time 415, 3581
DCMD_ALL_GETFLAGS 1253
DCMD_ALL_GETMOUNTFLAGS 1253
DCMD_ALL_SETFLAGS 1253
DEAD_PROCESS 995, 3633
DebugBreak() 416
debugging
kernel 418, 419
printing debugging messages (resolver routines) 2515
processes 416, 3123
shared objects 479
sockets 969
DebugKDBreak() 418
DebugKDOutput() 419
decimal digit, testing a character for 1443, 1472
decimal-point character, setting 2880
delay() 421
DELAYTIMER_MAX 3473
dev_t 2649
devctl() 424
  resource managers,
    implementing in 1253, 1257
DEVDIR_FROM 425
DEVDIR_NONE 425
DEVDIR_TO 425
DEVDIR_TOFROM 425
devices
  block special
    reading 2455
    writing 3794
character
  characters, injecting 3385
    size 3383, 3404
  classes of 2649
controlling 424, 1210
controlling device,
  making 3402
I/O memory, mapping 1685, 1873
mounts, autodetecting 1714
numbers
  attaching 2648
  detaching 2652
  getting 2558
  manipulating 3195
opening 1691
output, waiting for
  completion 3366
PCI
  attaching 2012
  configuration, reading 2040, 2043, 2045, 2048
    configuration, writing 2052, 2055, 2058, 2061
  detaching 2024
  finding 2026, 2029
  rescanning for 2050
  reading 1697, 2458
  script, running on 1701
  writing 1708
devp-pccard server
  arming 1994
  attaching 1997
  card insertion/removal,
    notification of 1994
CIS (Card Information Structure), reading 2007
  detaching 1999
  locking 2004
  socket setup information 2001
  unlocking 2008
diffime() 433
digit, testing a character for
  decimal 1443, 1472
  hexadecimal 1486, 1488
DIR 2464
dircntl() 435
direct memory access (DMA)
  channels, managing 2633
directories
  access, checking 126, 513
  base name 234
  closing 363
controlling 435
creating 1654, 1660
current working 309, 831, 1007
   for daemons 2106
deleting 2617
entries
   duplicate, filtering 435
   sorting 155
files, searching for 1984
hierarchy, walking 795, 1930
information, requesting 436
name 438
opening 1954
position
   getting 3409
   setting 2787
reading 2464, 2468
rewinding 2605
root 319
scanning 2674
dirent 2464, 2468
dirname() 438
dispatch_block() 441
dispatch_context_alloc() 445
dispatch_context_free() 448
dispatch_create() 450
dispatch_destroy() 453
dispatch_handler() 456
dispatch_t 451
dispatch_timeout() 460
dispatch_unblock() 463
dispatch interface See also
   resource managers
   blocking 441
   connections, creating 1646
   context
   allocating 445
   freeing 448
   endian, converting 1640
   events
   handling 456
   last selected 2803
   file descriptors
   handle, attaching 2796
   handle, detaching 2799
   handles
   creating 450
   destroying 453
   message handlers
   attaching 1640
   detaching 1649
   names
   attaching 1880
   detaching 1889
   server connections,
   closing 1887
   server connections,
   opening 1891
   path
   attaching to 2535
   detaching from 2554
   pulse handlers
   attaching 2380
   detaching 2383
   thread pool
   attributes, changing 3418,
   3431
   creating 3420
   destroying 3427
   starting 3433
   timeout, setting 460
   unblocking 463
div_t 465
Index

div() 465
division
terger 465
long integer 1515
DL_DEBUG 479
dl_info 467
dladdr() 468
dlclose() 470
dlerror() 472
dlopen() 474
dlsym() 481
DMA channels, managing 2633
dn_comp() 484
dn_expand() 486
domains
errors 1020, 1026, 1035
names
compressing 484
expanding 486
getting 372, 834
resolving 2515, 2519, 2522,
2525, 2528, 2530
setting 2845
secure RPC 372
UNIX 3616
dot notation (IP addresses) 1094
dotted quad (IP addresses) 1094
double-precision numbers
absolute value 257, 638
arccosines 129
arcsines 164
arctangents 196, 198
Bessel functions 1493, 1495,
1497, 3802, 3804, 3806
complementary error
function 563
cosines 398
cube roots 275
error function 561
exceptions, signal for 2989
exponentials 630, 632
exponents,
radix-independent 2665,
2668
finite, determining if 693
fractional part of a
double-precision number
1711
gamma functions 813, 1520
hyperbolic cosines 400
hyperbolic sines 3042
hyperbolic tangents 3364
hypotenuse, length of 1049
infinite, determining if 1449
input, formatted 2676
integral logarithms 1069
integral part of a
double-precision number
1711
integral power of 2 764, 1513
inverse hyperbolic cosines 131
inverse hyperbolic sines 166
inverse hyperbolic
tangents 200
logarithms 1547, 1549, 1551
modular arithmetic 707
next representable 1927
normalized fractions 764
not a number, determining
if 1453
powers 2092
precision 731
printing 2096
pseudo-random numbers 488, 559
radix-independent
    exponents 1553, 2665, 2668
remainders 490, 2507
residue 707
rounding 277, 703, 2612
sign, copying 396
significant bits 3012
sines 3040
square roots 3178
strings, converting
to/from 205, 3271
tangents 3362
times, difference between 433
wide-character strings,
converting to/from 3747
drand48() 488
drem(), dremf() 490
ds_clear() 492
ds_create() 495
ds_deregister() 497
ds_flags() 499
ds_get() 501
DS_PERM 494, 497, 499
ds_register() 503
ds_set() 505
dup() 507
dup2() 510
duplex connection, shutting
down 2952
dynamic rules filter
controlling 3540
dynamically linked libraries
addresses, translating 468
closing 470
debugging 479
erserrs 472
opening 474
symbol, getting address of 481

development and library functions
E
E2BIG 569
EACCES 569
eaccess() 513
EADDRINUSE 569
EADDRNOTAVAIL 569
EADV 569
EAFNOSUPPORT 569, 3106
EAGAIN 569
EALREADY 569
EALREADY 569
EBADF 569
EBADFD 569
EBADFSYS 569
EBADMSG 569
EBADR 569
EBADRQC 569
EBADSLT 569
EBFONT 569
EBUSY 569, 2005
ECANCELED 569
ECHILD 569
ECHO 3414
ECHOE 3414
ECHOK 3414
ECHONL 3414
ECHRN 569
ECOM 569
Index

ECONNABORTED 569
ECONNREFUSED 569, 1532
ECONNRESET 569
ECTRLTERM 569
EDEADLK 569
EDEADLOCK 569
EDESTADDRREQ 569
EDOM 569
EDQUOT 569
EEXIST 569
EFAULT 569
EFBIG 569
EHOSTDOWN 569
EHOSTUNREACH 569
EIDRM 569
EL2HLT 569
EL2NSYNC 569
EL3HLT 569
EL3RST 569
ELIBACC 569
ELIBBAD 569
ELIBEXEC 569
ELIBMAX 569
ELIBSCN 569
ELNRNG 569
ELPAGING 569
ELPUB 569
EMFILE 569
EMORE 569
EMPTY 3632
EMSGSIZE 569
EMULTIHOP 569
ENAMETOOLONG 569
Encapsulated Security Payload (ESP) 1406
encrypt() 517
encryption
key, setting 2875
passwords 406, 2414
strings 517
end-of-file
files 556
streams 666
clearing 328
endgrent() 520
endhostent() 521
endian
big

__BIGENDIAN__ manifest 113
little endian, converting
to/from 534, 536, 538, 540, 542, 544, 3291
messages 1640

native format, converting
to/from 522, 524, 526
unaligned values, accessing
safely 3603, 3605, 3607
unaligned values, writing
safely 3597, 3599, 3601
little

__LITTLEENDIAN__
manifest 113
big endian, converting
to/from 534, 536, 538, 540, 542, 544, 3291
messages 1640
native format, converting  
to/from  528, 530, 532
unaligned values, accessing  
safely  3603, 3605, 3607
unaligned values, writing  
safely  3597, 3599, 3601
ports  
reading from  1075, 1079
writing to  1969, 1973
ENDIAN_BE16()  522
ENDIAN_BE32()  524
ENDIAN_BE64()  526
ENDIAN_LE16()  528
ENDIAN_LE32()  530
ENDIAN_LE64()  532
ENDIAN_RET16()  534
ENDIAN_RET32()  536
ENDIAN_RET64()  538
ENDIAN_SWAP16()  540
ENDIAN_SWAP32()  542
ENDIAN_SWAP64()  544
endnetent()  546
endprotoent()  547
endpwent()  548
endservent()  550
endspent()  551
endutent()  553
ENETDOWN  569
ENETRESET  569
ENETUNREACH  569
ENFILE  569
ENOANO  569
ENOBUFS  569, 2622
ENOSCI  569
ENODATA  569
ENOENT  569
ENOEXEC  569
ENOLCK  569
ENOLIC  569
ENOLINK  569
ENOMEM  569
ENOMSG  569
ENONDPI  569
ENONET  569
ENOPKG  569
ENOPROTOOPT  569
ENOREMOTE  569
ENOSPC  569
ENOSR  569
ENOSTAT  569
ENOSYS  569
ENOTBLK  569
ENOTCONN  569
ENOTDIR  569
ENOTEMPTY  569
ENOTSOCK  569
ENOTSUP  569
ENOTTY  569
ENOTUNIQ  569
ENTER  1031
ENTRY  1031
env  598, 621, 2850, 3130, 3140,  
3157, 3167
environ  555, 1584, 2851
environment  
restoring  1558, 2979
saving  2872, 3025
environment variables  
COLUMNS  325

defining  555, 598, 620, 2394,  
2850, 3119, 3135, 3145,  
3150, 3162, 3171

September 10, 2007

Index  3925
deleting 325, 2394, 2850, 3119, 3135, 3145, 3150, 3162, 3171, 3622

**DL_DEBUG** 479
files, searching for 2782
getting 840

**HOSTALIASES** 871
**HOSTNAME** 881, 2865
**INCLUDE** 2782
**LD_LIBRARY_PATH** 479
**LIB** 2782
**LINES** 325
**LOCALDOMAIN** 2515, 2519, 2523, 2525, 2528, 2531
**PATH** 58, 325, 597, 620, 2782, 3139, 3144, 3149, 3166, 3170
pointer to 555

**SHELL** 325, 2076, 3357
**SNMPCONFIGFILE** 2453
**TERM** 325
**TERMINFO** 325
**TZ** 325, 3581

**ENXIO** 569
**EOF** 3656
**eof()** 556
**EOK** 569
**EOPNOTSUPP** 569, 3106
**EOVERFLOW** 569
**EPERM** 569
**EPFNOSUPPORT** 569
**EPIPE** 569
**EPROCNOSUPPORT** 569
**EPROGMISMATCH** 569
**EPROGMISMATCH** 569
**EPROGRANT** 569
**EPROGET** 569
**ERROR** 569
**ERANGE** 569
**EREMCHG** 569
**EREMOTE** 569
**ERESET** 569, 1770
**erf(), erff()** 561
**erfc()** 563
**erfcf()** 563
**EPROTONOSUPPORT** 569, 3106
**EPROTOTYPE** 569
**erand48()** 559
**ERANGE** 569
**EREMCHG** 569
**EREMOTE** 569
**ERESET** 569, 1770
**erf(), erff()** 561
**erfc()** 563
**erfcf()** 563
**EPROTONOSUPPORT** 569, 3106
**EPROTOTYPE** 569
**erand48()** 559

See also

- command-line options, printing for 907
- end-of-file 666
- **errno** global variable 568
- hosts 1020, 1026, 1035
- message-passing 1770
- messages for an error code 2065, 3227
- formatted 565, 3653
- regular expressions 2499
- resolver 1020, 1026, 1035
- signals, raising 2423
- socket address information 811
- **stderr** 69, 565, 2065, 3203, 3653
- stream I/O clearing 328
- testing for 668
escape characters 1708
ESHUTDOWN 569
ESOCKTNOSUPPORT 569
ESP (Encapsulated Security Payload) 1406
ESPIPE 569
ESRCH 569
ESRMNT 569
ESRVRFault 569
ESTALE 569
ESTRPIPE 569
ETIME 569
ETIMEDOUT 569
ETOOMANYREFS 569
ETXTBSY 569
EUNATCH 569
EUSERS 569

events
blocking while waiting for 441
channels, delivering through 1762
checking validity of 1854
handling 456
interrupts
attaching 1163
detaching 1169
last selected 2803

sigevent 2968
type, determining
withSIGEV_GET_TYPE() 2969
system
notification of 2109
triggering 2113
unblocking 463
user-generated 3540
EWOULDلنOACK 569, 973

exceptional conditions, file descriptors with 2632, 2790
exceptions, floating-point mask 725
registers 728
exclusive locks 699
EXDEV 569
exec* family of functions 50, 58, 1954
excl() 577
execl() 584
execlp() 591
execlpe() 597
executable files
base name 2122
file descriptor 367
full path 368
mounted filesystem, preventing from loading on 1713
execv() 601
execve() 607
execvp() 613
execvpe() 620
EXFULL 569
exit() 627
exp(), expf() 630
expm1(), expm1f() 632
exponentials, floating point 630, 632, 764, 1513
exponents, radix-independent 1553, 2665, 2668
export 598, 621, 2850, 3130, 3140, 3157, 3167
Index

© 2007, QNX Software Systems GmbH & Co. KG.

F

F_ALLOCSP 652, 1355
F_DUPFD 653
F_FREESP 653, 1355
F_GETFD 653
F_GETFL 653
F_GETLK 653
F_OK 126, 513
F_RDWR 654
F_SETFD 654
F_SETFL 654
F_SETLK 654
F_SETLKW 655
F_UNLOCK 654
F_WRLCK 654
fabs(), fabsf() 638
fast mode
controlling 3540
fchgetopen() 641
fchmod() 642
fchown() 645
FCHR_MAX 2907
fclose() 648
fcloseall() 650
fcntl() 652
F_DUPFD 507, 510
resource managers,
 implementing in 1276
FD See file descriptors
FD_CLOEXEC 654, 2944, 3118, 3149
FD_CLR() 2791
FD_ISSET() 2791
FD_SET() 2791
FD_SETSIZE 2791
FD_ZERO() 2791

fdatasync() 661
fdinfo()
 resource managers,
 implementing in 1260, 1261
fdopen() 663
feof() 666
ferror() 668
fflush() 670
fget() 672
fgetc() 673
fgetchar() 675
fgetpos() 677
fgets() 679, 954
fgetspent() 682
fgetwc() 685
fgetws() 687
FIFO scheduling 2161
FIFOs
 creating 1657, 1660
 reading from 2094, 2446, 2474
 unnamed (pipes)
 closing 2063
 creating 2067
 opening 2076
file descriptors
 closing in a child
 process 3118, 3149
 connection IDs as 381
 creating 1947, 3111
 duplicating 507, 510, 653, 1957, 3116
 exceptional conditions 2632, 2790
 full path 3576, 3578
 handle
 attaching 2796
detaching 2799
maximum number of 2903
polling 2069
properties 1445
ready for reading or
writing 2632, 2790
selecting 2632, 2790
sets of, manipulating 2791
stderr 3203
stdin 3204
stdout 3205
streams, associating with 663,
690
table size, getting 836
terminal, testing for association
with 1439

fileno() 70, 690
files
_PATH_UTMP 553
_PATH_UTMP 993, 996,
998, 2403, 2924, 3634
access times 3629
disabling logging of on
mounted filesystems
1713
resource managers,
implementing in 1370,
1382, 1385
access, checking 126, 513
base name 234
closing 361
configurable limits 736, 1336,
1980
configuration, opening 286,
641
controlling 652
core, maximum size 2903
creating 714, 1947, 3111
low-level 402
not allowing on mounted
filesystems 1713
resource manager 1200,
1322
deleting 2509, 3619
device parameters,
manipulating 1210
directory name 438
executable
base name 2122
file descriptor 367
full path 368
extending 322, 652
flushing 670
all 705
information, getting 773,
1570, 3191
input, formatted 766, 809,
3660, 3665
link count, maximum 736,
1980
linking to 695, 1523, 1987,
1989, 2471, 3297
locking 653, 698, 1544
by a thread 701
nonblocking 792
modification times 799, 3626,
3629
resource managers,
implementing in 1370,
1382, 1385
names
full path 3576, 3578
matching 711
maximum length 736, 1980
opening 402, 714, 1947, 3111
resource manager 1322
output, formatted 739, 804, 3657, 3663
ownership, changing 316, 645, 1508
pathnames matching a pattern 1010, 1014
permissions
  changing 312, 642
daemons 2106
  on creation 3592
  restricting the changing of 737, 1981
position
  getting 677
  setting 769, 771, 1567, 3406
private access 1957
processes, maximum files per 3335
reading 1151, 2094, 2446
  blocks 2455
  characters, number waiting to be read 3388
  checking 2440
  iov_t 2474
renaming 2512
reopening 760
rewinding 2602
scanning directories for 2674
searching
  environment variables 2782
  list of directories 1984
seeking 769
shared access 3111, 3116
size, changing 322
size, maximum 2903
SNMP configuration 2451
status flags 653
status-change times
  resource managers, implementing in 1370, 1382, 1385
synchronizing 661, 781
temporary
  creating 3522
  creating and opening 1664
  name, generating 1666, 3411, 3525
tree, walking 795, 1930
truncating 322, 653, 789, 1576, 3573
unlinking 2509, 3619
unlocking 654, 698, 797, 1544
writing 2411, 3789
  blocks 3794
  characters 741, 743
  iov_t 3797
  strings 745
  wide characters 747
  wide-character strings 749
filesystems
  free space 778
  information, getting 777, 3199
  mounting 1714
    options, parsing 1716
    read-only 1713
    synchronizing 3300
    requesting 2113
    unmounting 1714, 3595
filters
  controlling 3540
FIND 1031
finite number, determining if 693
finite(), finitef() 693
first-in first-out scheduling 2161
flink() 695
floating point
absolute value 257, 638
arccosines 129
arcsines 164
arctangents 196, 198
Bessel functions 1493, 1495, 1497, 3802, 3804, 3806
complementary error function 563
cosines 398
cube roots 275
error function 561
exceptions, signal for 2989
exponentials 630, 632
exponents, radix-independent 2665, 2668
finite, determining if 693
fractional part 1711
gamma functions 813, 1520
hyperbolic cosines 400
hyperbolic sines 3042
hyperbolic tangents 3364
hypotenuse, length of 1049
infinite, determining if 1449
input, formatted 2676
integer logarithms 1069
integral part 1711
integral power of 2 764, 1513
inverse hyperbolic cosines 131
inverse hyperbolic sines 166
inverse hyperbolic tangents 200
logarithms 1547, 1549, 1551
modular arithmetic 707
next representable 1927
normalized fractions 764
not a number, determining if 1453
powers 2092
printing 2096
radix-independent exponents 1553, 2665, 2668
remainders 490, 2507
residue 707
rounding 277, 703, 2612
settings
exception mask 725
exception registers 728
precision 731
rounding 734
sign, copying 396
significant bits 3012
sines 3040
square roots 3178
tangents 3362
wide-character strings, converting to/from 3747
flock 655
flock() 698
flockfile() 701
floor(), floorf() 703
flow control 3371
flushall() 705
fmod(), fmodf() 707
FNMPATHNAME 710
FNM_PERIOD 710
FNMQQUOTE 710
fnmatch() 711
fopen() 714
fork()  719, 1954
forkpty()  723
fp_exception_mask()  725
fp_exception_value()  728
fp_precision()  731
fp_rounding()  734
fpathconf()  736
fpos_t  677, 771
fprintf()  739
fputc()  741
fputchar()  743
fputs()  745
fputwc()  747
fputws()  749
FQNN (Fully Qualified Node Name)  1919
FQPN (Fully Qualified Path Name)  1918
fractional part of a floating-point number  1711
fread()  751
free memory, amount of  3197
free space, filesystem  778, 3200
free()  754
freeaddrinfo()  756
freeaddrinfo()  758
freopen()  70, 760
frewpt(), frexpfp()  764
fscanf()  766
fseek()  769, 2602
fsetpos()  771
fstat(), fstat64()  773
fstatvfs(), fstatvfs64()  777
fstruncate(), fstruncate64()  789
ftrylockfile()  792
FTW_D  795, 1931
FTW_DNR  795, 1931
FTW_F  795, 1931
FTW_NS  795, 1932
ftw()  795
full-duplex connection, shutting down  2952
Fully Qualified Node Name (FQNN)  1919
Fully Qualified Path Name (FQPN)  1918
function classification  105
safety  109
funlockfile()  797
futime()  799
fwrite()  806
fwide()  802
fwprintf()  804
fwrite()  806
fscanf()  809

G

gai_strerror()  811
gamma functions  813, 1520
gamma(), gamma_r(), gammaf(), gammas_r()  813
General Purpose Interface Bus (GPIB)  1993
get_device_command()  425
get_device_direction()  425
GET_REQ_MSG  3080
GET_RSP_MSG  3080
getaddrinfo()  816
getc_unlocked() 825
getc() 823
getchar_unlocked() 829
getchar() 827
getcwd() 831
getdomainname() 834
getdtablesize() 836
getegid() 838
getenv() 598, 621, 840, 3130, 3140, 3157, 3167
geteuid() 842
getgid() 844
getgrent() 846
getgrgid_r() 851
getgrgid() 849
getgrnam_r() 856
getgrnam() 854
getgrouplist() 860
getgroups() 862
gethostbyaddr_r() 868
gethostbyaddr() 864
gethostbyname_r() 874
gethostbyname(), gethostbyname2() 870
gethostent_r() 879
gethostent() 876
gethostname() 881
getifaddrs() 883
GETIOVBASE() 885
GETIOVLEN() 887
getitimer() 889
getlogin_r() 893
getlogin() 891
getnameinfo() 895
getnetbyaddr() 900
getnetbyname() 902
getnetent() 904
GETNEXT_REQ_MSG 3080
getopt() 906
getpass() 912
getpeername() 914
getpgid() 916
getpgrp() 918
getpid() 920
getppid() 922
getprior() 924
getprotobynumber() 928
getprotoent() 930
getpwent() 932
getpwnam_r() 938
getpwnam() 935
getpwuid_r() 943
getpwuid() 940
getrusage() 949
getrusage(), getrusage64() 946
getruent() 954
getservbyname() 956
getservbyport() 958
getservent() 960
getsocknam() 962
getsockopt() 967
getspent() 977
getspnam(), getspnam_r() 981
getspnam() 984
gettimeofday() 989
getuid() 991
getutent() 993
getutid() 995
getutline() 998
getw() 1001
getwc() 1003
getwchar() 1005
getwd() 1007

`glob_t` 1009

glob() 1010

global variables

  __progname 2122
  __amblksiz 157
  __argc 159
  __argv 160
  __auxv 233
  __btext 252
  __edata 516
  __end 519
  __etext 576
  __syspage_ptr 3355

daylight 415
erro 568

optarg 906

opterr 907

optind 906

optopt 907

stderr 3203

stdin 3204

stdout 3205

sys_errnolist 569

sys_nerr 569

sys_nsig 2988

sys_siglist 2988

timezone 3519

tzname 3580

globfree() 1014

gmtime_r() 1018

gmtime() 1016

GPIB (General Purpose Interface Bus) 1993

greater of two numbers 1595

groups  See also process groups

    access list

    getting 860

    initializing 1146

database

    closing 520

    membership 1146

    next entry, getting 846

    rewinding 2859

IDs

    effective 838, 2847, 2856,

    2896

    information about 849, 851

    process 918

    real 844, 2856, 2896

    saved 2856

    name, getting information

    about 854, 856

    set-group ID 3335

    supplementary

    IDs 862, 2861

    maximum per process 3334

    guard area 2132, 2155

    guardian process, specifying 2115

H

    `h_errno` 865, 871, 1020, 1026,

    2522, 2528

    hardware

    information in system

    page 1042, 1044, 1045,

    1047

    manufacturer, getting 372

    serial number, getting 372

    type 372, 3609

    hardware interrupts  See interrupts
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash table</td>
<td></td>
</tr>
<tr>
<td>creating</td>
<td>1022</td>
</tr>
<tr>
<td>destroying</td>
<td>1024</td>
</tr>
<tr>
<td>searching</td>
<td>1031</td>
</tr>
<tr>
<td>hcreate()</td>
<td>1022</td>
</tr>
<tr>
<td>hdestroy()</td>
<td>1024</td>
</tr>
<tr>
<td>herror()</td>
<td>1026</td>
</tr>
<tr>
<td>hexadecimal numbers</td>
<td></td>
</tr>
<tr>
<td>digit, testing a character</td>
<td>1486, 1488</td>
</tr>
<tr>
<td>strings, converting to/from</td>
<td>207</td>
</tr>
<tr>
<td>holding a process for debugging</td>
<td>3123</td>
</tr>
<tr>
<td>HOST_NOT_FOUND</td>
<td>1020, 1026</td>
</tr>
<tr>
<td>host-byte order</td>
<td></td>
</tr>
<tr>
<td>network-byte order, converting to/from</td>
<td>1037, 1039, 1940, 1942</td>
</tr>
<tr>
<td>HOSTALIASES</td>
<td>871</td>
</tr>
<tr>
<td>hostent</td>
<td>1029</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>881, 2865</td>
</tr>
<tr>
<td>hosts</td>
<td></td>
</tr>
<tr>
<td>addresses</td>
<td></td>
</tr>
<tr>
<td>strings, converting to/from</td>
<td>1106, 1109</td>
</tr>
<tr>
<td>database</td>
<td></td>
</tr>
<tr>
<td>closing</td>
<td>521</td>
</tr>
<tr>
<td>entries, getting</td>
<td>864, 868, 870, 874, 876, 879</td>
</tr>
<tr>
<td>errors</td>
<td>1020, 1026, 1035</td>
</tr>
<tr>
<td>opening</td>
<td>2863</td>
</tr>
<tr>
<td>structure</td>
<td>1029</td>
</tr>
<tr>
<td>domain name</td>
<td></td>
</tr>
<tr>
<td>getting</td>
<td>372, 834</td>
</tr>
<tr>
<td>setting</td>
<td>2845</td>
</tr>
<tr>
<td>names</td>
<td>881, 2865</td>
</tr>
<tr>
<td>getting</td>
<td>372</td>
</tr>
<tr>
<td>valid characters</td>
<td>372</td>
</tr>
<tr>
<td>remote</td>
<td></td>
</tr>
<tr>
<td>identity, checking</td>
<td>2657</td>
</tr>
<tr>
<td>hot swapping</td>
<td>2050</td>
</tr>
<tr>
<td>hsearch()</td>
<td>1031</td>
</tr>
<tr>
<td>hstrerror()</td>
<td>1035</td>
</tr>
<tr>
<td>htonl()</td>
<td>1037</td>
</tr>
<tr>
<td>htons()</td>
<td>1039</td>
</tr>
<tr>
<td>hwifind_item()</td>
<td>1042</td>
</tr>
<tr>
<td>hwifind_tag()</td>
<td>1044</td>
</tr>
<tr>
<td>HWI_NULL_OFF</td>
<td>1042</td>
</tr>
<tr>
<td>hwioff2tag()</td>
<td>1045</td>
</tr>
<tr>
<td>hwitag2off()</td>
<td>1047</td>
</tr>
<tr>
<td>hyperbolic functions</td>
<td></td>
</tr>
<tr>
<td>trigonometry</td>
<td></td>
</tr>
<tr>
<td>hyperbolic cosine</td>
<td>400</td>
</tr>
<tr>
<td>hyperbolic sine</td>
<td>3042</td>
</tr>
<tr>
<td>hyperbolic tangent</td>
<td>3364</td>
</tr>
<tr>
<td>inverse hyperbolic cosine</td>
<td>131</td>
</tr>
<tr>
<td>inverse hyperbolic sine</td>
<td>166</td>
</tr>
<tr>
<td>inverse hyperbolic tangent</td>
<td>200</td>
</tr>
<tr>
<td>hypot(), hypotf()</td>
<td>1049</td>
</tr>
<tr>
<td>hypotenuse, length of</td>
<td>1049</td>
</tr>
<tr>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>buffers, flushing</td>
<td>3374</td>
</tr>
<tr>
<td>configuration files, opening</td>
<td>286</td>
</tr>
<tr>
<td>end-of-file, checking for</td>
<td>556</td>
</tr>
<tr>
<td>FIFOs, creating</td>
<td>1657, 1660</td>
</tr>
<tr>
<td>file descriptors</td>
<td></td>
</tr>
</tbody>
</table>
Index

duplicating 507, 510, 653, 1957, 3116
selecting 2632, 2790
file-mode creation mask 3592
daemons 2106
files
  closing 361
  controlling 652
  extending 652
  information, getting 773
  linking to 1523, 1987, 1989, 2471, 3297
  locking 653, 1544
  names, matching 711
  opening 402, 1947, 3111
  reading 2094, 2446, 2455
  status flags 653
  truncating 653
  unlocking 654, 698, 1544
  writing 2411, 3789, 3794
filesystems
  information, getting 777, 3199
iov_t
  reading 2474
  writing 3797
ports, managing 2633
privileges, requesting 1155, 1164, 1169, 1182, 1184, 1187, 1189, 3449
requests, initiating list of 1528
I/O functions (resource managers)
  chmod 1232
    default 1235
  chown 1237
    default 1240
  close
    default 1244, 1247
    default values, setting 1265
devctl 1253
  default 1257
  fdinfo 1260
  default 1261
  link 1268
  lock
    default 1276
  lseek 1283
    default 1286
  mknod 1289
  mmap 1291
  default 1295
  openfd 1330
  default 1334
  pathconf 1336
  default 1339
  read 1344
  default 1341
  readdir 1347
  space 1355
  stat 1358
  default 1360
  sync 1367
  default 1365
  utime 1382
  default 1385
  write
    default 1388
I/O vector
  base, getting 885
  fields, filling 2867
  length, getting 887
  reading from a file 2474
  writing to a file 3797
ICANON 3414
ICMP (Internet Control Message Protocol) 1054
ICMP6 (Internet Control Message Protocol v6) 1056
ICMP6_FILTER 1056
ICMP6_FILTER_SETBLOCK() 1057
ICMP6_FILTER_SETBLOCKALL() 1057
ICMP6_FILTER_SETPASS() 1057
ICMP6_FILTER_SETPASSALL() 1057
ICMP6_FILTER_WILLBLOCK() 1057
ICMP6_FILTER_WILLPASS() 1057
ICRNL 3413
IEXTEN 3415
if_freenameindex() 1059
if_indextoname() 1061
if_msghdr 2623
if_nameindex 1063
if_nameindex() 1063
if_nametoindex() 1065
ifa_msghdr 2623
ifaddr 1067
IFNAMSIZ 1061
IGNBRK 3413
IGNCR 3413
IGNPAR 3413
IHFLOW 3414
ilogb(), ilogbff() 1069
in16() 1075
in16s() 1077
in32() 1079
in32s() 1081
in6_pktinfo 1427
in8() 1071
in8s() 1073
INADDR_ANY 1402, 3390
INADDR_NONE 1085, 1086, 1100
inbe16() 1075
inbe32() 1079
INCLUDE 2782
incoming connections, listening
index() 1083
inet_addr() 1085
INET_ADDRSTRLEN 1106
inet_aton() 1087
inet_inaof() 1089
inet_makeaddr() 1091
inet_net_ntop() 1093
inet_net_pton() 1098
inet_netof() 1096
inet_network() 1100
inet_ntoa_r() 1104
inet_ntoa() 1102
inet_ntop() 1106
inet_pton() 1109
INET6 (Internet protocol v6 family) 1114
INET6_ADDRSTRLEN 1106
inet6_option_alloc() 1118, 1120, 1122, 1124, 1126, 1128
inet6_option_append() 1118, 1120, 1122, 1124, 1126, 1128
inet6_option_find() 1118, 1120, 1122, 1124, 1126, 1128
inet6_option_init() 1118, 1120, 1122, 1124, 1126, 1128
inet6_option_next() 1118, 1120, 1122, 1124, 1126, 1128
inet6_option_space() 1118, 1120, 1122, 1124, 1126, 1128
inet6_rthdr_add() 1130
inet6_rthdr_getaddr() 1132
inet6_rthdr_getflags() 1134
inet6_rthdr_init() 1136
inet6_rthdr_lasthop() 1138
inet6_rthdr_reverse() 1140
inet6_rthdr_segments() 1142
inet6_rthdr_space() 1144
infinite number, determining if 1449
INFORM_REQ_MSG 3080
inheritance 3122
INIT_PROCESS 995, 3632
initgroups() 1146
initstate() 1148
INLCR 3413
inle16() 1075
inle32() 1079
inodes
  getting 2558
  number of 778, 3200
INPCK 3413
input_line() 1151
input, formatted 766, 809, 2676, 3660, 3665, 3669, 3693, 3800
instruction set architecture 371
instrumented kernel 1180
integers
  absolute value 121
  atomic operations
    addition 213, 215
    subtraction 225, 227
division 465
pseudo-random numbers 2426, 2428
quotient 465
remainder 465
rounding 277, 703
size of 113
system message log, writing to 3066
integral logarithms 1069
integral part of a floating-point number 1711
integral power of 2 764, 1513
Intel 80x86-specific interrupts 1194
interfaces
  index, mapping to name 1061
  list of, freeing 1059
  list of, getting 1063
  name, mapping to index 1065
international characters See wide characters
Internet Control Message Protocol
  See ICMP
Internet domain
  errors 1020, 1026, 1035
  name servers
    initializing 2515
    querying 2519, 2522, 2525, 2528, 2530
  names
    compressing 484
    expanding 486
Internet Protocol
  See IP
  InterruptAttach(), InterruptAttach_r() 1155
InterruptAttachEvent(),
  InterruptAttachEvent_r()  1163
InterruptDetach(),
  InterruptDetach_r()  1169
InterruptDisable()  1172
InterruptEnable()  1174
InterruptHookIdle()  1176
InterruptHookTrace()  1180
InterruptLock()  1182
InterruptMask()  1184
interrupts
classes 1155
disabling 1172
enabling 1174
events
  attaching 1163
detaching 1169
handlers
  attaching 1155
detaching 1169
dispersing hardware
  interrupts 1172
guidelines for writing 1158
idle, attaching 1176
locking 1182, 1187
stack size 1158
Intel 80x86-specific 1194
level-sensitive 1184
masking 1184, 1189
PCI
  mapping 2035
  routing information 2032
  requests, managing 2633
  waiting for 1191
InterruptUnlock()  1187
InterruptUnmask()  1189
InterruptWait(), InterruptWait_r()  1191
inverse hyperbolic cosines 131
inverse hyperbolic sines 166
inverse hyperbolic tangents 200
io_chmod_t 1233
io_chown_t 1238
io_close_t 1245
IO_CONNECT 1198
io_devctl_t 1254
io_fdinfo_t 1262
io_link_t 1269
io_lseek_t 1284
io_mknod_t 1289
io_mmap_t 1292
io_notify_t 1299
io_open_t 1324
io_openfd_t 1331
io_pathconf_t 1337
io_read_t 1344
io_readlink_t 1348
io_rename_t 1351
io_space_t 1355
io_stat_t 1361
io_sync_t 1367
io_unlink_t 1378
io_utime_t 1383
io_write_t 1391
ioctl() 1210
iofdinfo() 1212
IOFUNC_ATTR_ATIME 1220, 1370, 1386
IOFUNC_ATTR_CTIME 1220, 1370, 1386
IOFUNC_ATTR_DIRTY_MODE 1220, 1386
iofunc_space_verify()  1355
iofunc_stat_default()  1360
iofunc_stat()  1358
iofunc_sync_default()  1365
iofunc_sync_verify()  1367
iofunc_sync()  1363
iofunc_time_update()  1370
iofunc_unblock_default()  1374
iofunc_unblock()  1372
iofunc_unlink()  1377
iofunc_unlock_ocb_default()  1380
iofunc_utime_default()  1385
iofunc_utime()  1382
iofunc_write_default()  1388
iofunc_write_verify()  1391
ionotify()  1394

io_t  2474, 2867, 3797
  base, getting  885
  fields, filling  2867
  length, getting  887
  reading from a file  2474
  resource managers  2586
  writing to a file  3797

iovec  1631
  copying
IP (Internet Protocol)  1400
  IP_ADD_MEMBERSHIP  1403
  IP_DROP_MEMBERSHIP  1404
  IP_HDRINCL  968, 1401
  IP_MAX_MEMBERSHIPS  1404
  ip_mreq  1404
  IP_MULTICAST_IF  1402
  IP_MULTICAST_LOOP  1403
  IP_MULTICAST_TTL  1402
  IP_OPTIONS  1400
  IP_RECVSTADDR  1401
  IP_RECVIF  1401
  IP_TOS  968, 1400
  IP_TTL  1400

IP addresses
  CIDR (Classless Internet Domain Routing),
    converting to/from  1093, 1098
  classes of  1094
  extracting
    network number  1096
  local network addresses,
    extracting  1089
  manipulating  1091
  network numbers, converting
    to/from strings  1100
  specifying in dot notation  1094
  strings, converting
    to/from  1085, 1087, 1102,
    1104, 1106, 1109
IP Payload Compression
  Protocol  1411
IP6 (Internet Protocol v6)  1423
IPPROTO_ICMPV6  1056
IPPROTO_RAW  1401, 1430
IPsec (secure Internet
  Protocol)  1406
  errors  1421
  policies  1413, 1415, 1417
    ipsec_dump_policy()  1413
    ipsec_get_policylen()  1415
    ipsec_set_policy()  1417
    ipsec_strerror()  1421
IPv6
  hop-by-hop and destination
    options  1118, 1120, 1122,
    1124, 1126, 1128
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>router header options</td>
<td>1136, 1144</td>
</tr>
<tr>
<td>IPV6_BINDV6ONLY</td>
<td>1426</td>
</tr>
<tr>
<td>IPV6_DSTOPTS</td>
<td>1426</td>
</tr>
<tr>
<td>IPV6_HOPLIMIT</td>
<td>1426</td>
</tr>
<tr>
<td>IPV6_HOPOPTS</td>
<td>1426</td>
</tr>
<tr>
<td>IPV6_JOIN_GROUP</td>
<td>1425</td>
</tr>
<tr>
<td>ipv6_mreq</td>
<td>1425</td>
</tr>
<tr>
<td>IPV6_MULTICAST_HOPS</td>
<td>1423</td>
</tr>
<tr>
<td>IPV6_MULTICAST_IF</td>
<td>1424</td>
</tr>
<tr>
<td>IPV6_MULTICAST_LOOP</td>
<td>1424</td>
</tr>
<tr>
<td>IPV6_PKTINFO</td>
<td>1426</td>
</tr>
<tr>
<td>IPV6_PORTRANGE</td>
<td>1425</td>
</tr>
<tr>
<td>IPV6_RTHDR</td>
<td>1426</td>
</tr>
<tr>
<td>IPV6_UNICAST_HOPS</td>
<td>1423</td>
</tr>
<tr>
<td>IRQs (Interrupt ReRequests), managing</td>
<td>2633</td>
</tr>
<tr>
<td>isalnum()</td>
<td>1433</td>
</tr>
<tr>
<td>isalpha()</td>
<td>1435</td>
</tr>
<tr>
<td>isascii()</td>
<td>1437</td>
</tr>
<tr>
<td>isatty()</td>
<td>1439</td>
</tr>
<tr>
<td>iscntrl()</td>
<td>1441</td>
</tr>
<tr>
<td>isdigit()</td>
<td>1443</td>
</tr>
<tr>
<td>isfdtype()</td>
<td>1445</td>
</tr>
<tr>
<td>isgraph()</td>
<td>1447</td>
</tr>
<tr>
<td>ISIG</td>
<td>3415</td>
</tr>
<tr>
<td>isinf(), isnanf()</td>
<td>1449</td>
</tr>
<tr>
<td>islower()</td>
<td>1451</td>
</tr>
<tr>
<td>isnan(), isnanf()</td>
<td>1453</td>
</tr>
<tr>
<td>isprint()</td>
<td>1455</td>
</tr>
<tr>
<td>ispunct()</td>
<td>1457</td>
</tr>
<tr>
<td>isspace()</td>
<td>1459</td>
</tr>
<tr>
<td>ISTRIP</td>
<td>3413</td>
</tr>
<tr>
<td>isupper()</td>
<td>1462</td>
</tr>
<tr>
<td>iswalnum()</td>
<td>1464</td>
</tr>
<tr>
<td>iswalpha()</td>
<td>1466</td>
</tr>
<tr>
<td>iswctype()</td>
<td>1470</td>
</tr>
<tr>
<td>iswdigit()</td>
<td>1472</td>
</tr>
<tr>
<td>iswgraph()</td>
<td>1474</td>
</tr>
<tr>
<td>iswlower()</td>
<td>1476</td>
</tr>
<tr>
<td>iswprint()</td>
<td>1478</td>
</tr>
<tr>
<td>iswpunct()</td>
<td>1480</td>
</tr>
<tr>
<td>iswspace()</td>
<td>1482</td>
</tr>
<tr>
<td>iswupper()</td>
<td>1484</td>
</tr>
<tr>
<td>isxdigit()</td>
<td>1488</td>
</tr>
<tr>
<td>ITIMER_REAL</td>
<td>890, 2870</td>
</tr>
<tr>
<td>itimerspec</td>
<td>3475</td>
</tr>
<tr>
<td>itoa()</td>
<td>1490</td>
</tr>
<tr>
<td>IFFPOFF</td>
<td>3413</td>
</tr>
<tr>
<td>IXON</td>
<td>3413</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>j0(), j0f()</td>
<td>1493</td>
</tr>
<tr>
<td>j1(), j1f()</td>
<td>1495</td>
</tr>
<tr>
<td>jn(), jnf()</td>
<td>1497</td>
</tr>
<tr>
<td>job control, supporting</td>
<td>3335</td>
</tr>
<tr>
<td>jrand48()</td>
<td>1499</td>
</tr>
<tr>
<td>jumps, nonlocal</td>
<td>1558, 2872, 2979, 3025</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>kerinfo</td>
<td>3350</td>
</tr>
<tr>
<td>kernel</td>
<td></td>
</tr>
<tr>
<td>blocking states, setting timeouts</td>
<td>3506</td>
</tr>
<tr>
<td>calls</td>
<td></td>
</tr>
</tbody>
</table>
Index

asynchmsg_connect_attach() 177
asynchmsg_connect_attr() 179
asynchmsg_flush() 185
asynchmsg_connect_detach() 181
asynchmsg_free() 187
asynchmsg_get() 189
asynchmsg_malloc() 191
ChannelCreate(),
  ChannelCreate_r() 300
ChannelDestroy(),
  ChannelDestroy_r() 307
ClockAdjust(),
  ClockAdjust_r() 347
ClockCycles() 349
ClockId(), ClockId_r() 351
ClockPeriod(),
  ClockPeriod_r() 355
ClockTime(), ClockTime_r() 358
ConnectAttach(),
  ConnectAttach_r() 380
ConnectClientInfo(),
  ConnectClientInfo_r() 384
ConnectDetach(),
  ConnectDetach_r() 303, 388
ConnectFlags(),
  ConnectFlags_r() 390
ConnectServerInfo(),
  ConnectServerInfo_r() 393
DebugBreak() 416
DebugKDBreak() 418
DebugKDOoutput() 419
InterruptAttach(),
  InterruptAttach_r() 1155
InterruptAttachEvent(),
  InterruptAttachEvent_r() 1163
InterruptDetach(),
  InterruptDetach_r() 1169
InterruptHookIdle() 1176
InterruptHookTrace() 1180
InterruptMask() 1184
InterruptUnmask() 1189
InterruptWait(),
  InterruptWait_r() 1191
MsgDeliverEvent(),
  MsgDeliverEvent_r() 1762
MsgError(), MsgError_r() 1770
MsgInfo(), MsgInfo_r() 1773
MsgKeyData(),
  MsgKeyData_r() 1776
MsgRead(), MsgRead_r() 1783
MsgReadv(), MsgReadv_r() 1787
MsgReceive(),
  MsgReceive_r() 1791
MsgReceivePulse(),
  MsgReceivePulse_r() 1797
MsgReceivePulsev(),
  MsgReceivePulsev_r() 1801
MsgReceivev(),
  MsgReceivev_r() 1805
Index

MsgReply(), MsgReply_r() 1809
MsgReplyv(), MsgReplyv_r() 1812
MsgSend(), MsgSend_r() 1816
MsgSendnc(),
    MsgSendnc_r() 1821
MsgSendPulse(),
    MsgSendPulse_r() 1826
MsgSendsv(),
    MsgSendsv_r() 1830
MsgSendsvnc(),
    MsgSendsvnc_r() 1834
MsgSendv(), MsgSendv_r() 1838
MsgSendvnc(),
    MsgSendvnc_r() 1842
MsgSendsv(),
    MsgSendsv_r() 1846
MsgSendsvnc(),
    MsgSendsvnc_r() 1850
MsgVerifyEvent(),
    MsgVerifyEvent_r() 1854
MsgWrite(), MsgWrite_r() 1856
MsgWritev(), MsgWritev_r() 1860
SchedCtl(), SchedCtl_r() 2713
SchedGet(), SchedGet_r() 2747
SchedInfo(), SchedInfo_r() 2751
SchedSet(), SchedSet_r() 2754
SchedYield(), SchedYield_r() 2757
SignalAction(),
    SignalAction_r() 2988
SignalKill(), SignalKill_r() 2996
SignalProcmask(),
    SignalProcmask_r() 3002
SignalSuspend(),
    SignalSuspend_r() 3006
SignalWaitinfo(),
    SignalWaitinfo_r() 3009
SyncCondvarSignal(),
    SyncCondvarSignal_r() 3301
SyncCondvarWait(),
    SyncCondvarWait_r() 3304
SyncCtl(), SyncCtl_r() 3310
SyncDestroy(),
    SyncDestroy_r() 3312
SyncMutexEvent(),
    SyncMutexEvent_r() 3314
SyncMutexLock(),
    SyncMutexLock_r() 3317
SyncMutexRevive(),
    SyncMutexRevive_r() 3320
SyncMutexUnlock(),
    SyncMutexUnlock_r() 3322
SyncSemPost(),
    SyncSemPost_r() 3325
SyncSemWait(),
    SyncSemWait_r() 3327
SyncTypeCreate(),
    SyncTypeCreate_r() 3331
ThreadCancel(),
    ThreadCancel_r() 3436
ThreadCreate(),
    ThreadCreate_r() 3441
ThreadCtl(), ThreadCtl_r()
    1155, 1164, 1169, 1172,
    1174, 1182, 1184, 1187,
    1189, 3447
ThreadDestroy(),
    ThreadDestroy_r() 3455
ThreadDetach(),
    ThreadDetach_r() 3458
ThreadJoin(),
    ThreadJoin_r() 3460
TimerAlarm() 3489
TimerCreate() 3492
TimerDestroy(),
    TimerDestroy_r() 3495
TimerInfo(), TimerInfo_r()
    3498
TimerSettime(),
    TimerSettime_r() 3502
TimerTimeout(),
    TimerTimeout_r() 3506
debbuging 418, 419
instrumented 1180
thread scheduler 2751
kill() 1501
killpg() 1504

L

L_ctermid 408

L_tmpnam 3525
labs() 1506
large-file support 105
file information 773, 1570, 3191
filesystem information 777, 3199
mapped memory, offset of 1618
opening 402, 1947
position setting 1567, 3406
reading 2094
shared memory, mapping 1679
symbolic link information 1570
system-resource limits 946, 2902
truncating 789
writing 2411

LC_ALL 2879
LC_COLLATE 2879
LC_CTYPE 1471, 2879
LC_MESSAGES 2879
LC_MONETARY 2879
LC_NUMERIC 2879
LC_TIME 2879
lchown() 1508
lcong48() 1511
lconv 1534

LD_LIBRARY_PATH 479
ldexp(), ldexpf() 1513
ldiv_t 1515
ldiv() 1515
length, calculating
    hypotenuse 1049
    strings 3237
Index

wide-character strings 3729
lesser of two numbers 1652
level-sensitive interrupts 1184
find() 1517
lgamma(), lgamma_r(), lgammaf(),
   lgammaf_r() 1520
LIB 2782
libraries
   linking against 104
   locating 372
limits
   core files, size of 2903
   data segment, size of 2903
   device numbers 2649
files
   descriptors, number of 2903
   link count 736, 1980
   maximum per process 3335
   names, length of 736, 1980
size 2903
filesystems 777, 3199
hops 1423, 1426
host names, length 2865
inheriting 580
iov arrays 2474
memory locked into physical
   memory 2903
path names, length of 737, 1981
pipes, number of bytes written
   atomically 737, 1981
processes
   argument lists 3334
   CPU time 2903
   execution time 2702
   files, number open 3335
   I/O requests 1527
mapped address space 2903
maximum per real user
   ID 2903, 3334
   scheduling policy 2690, 2692
   supplementary group
   IDs 3334
resident set size 2903
sockets, pending
   connections 1532, 2615
stack size 2903
system resources
   getting 836, 946
   setting 2902
TCP maximum segment
   size 3391
terminals
   canonical input buffer
   size 736, 1980
   raw input buffer size 736, 1980
threads
   execution time 2751
   maximum number of 2903
   priority 2751
   stack size 2171
line buffering, setting for stream
   I/O 2877
linear search 1517, 1563
LINES 325
LINK_MAX 696, 1524
link() 1523
   resource managers,
   implementing in 1268
link-local addresses 1115
linker symbols
   _btext 252
Index

_edata 516
_end 519
_etext 576

links, symbolic
creating 3297
deleting 2509, 3619
information, getting 1570
ownership, changing 1508
reading 2471
resolving 2481
temporary 1987, 1989

lio_listio() 1528
LIO_NOP 1528
LIO_NOWAIT 1527
LIO_READ 1528
LIO_WAIT 1527
LIO_WRITE 1528
listen() 1532, 3390, 3813

little endian
__LITTLEENDIAN__ manifest 113
big endian, converting
to/from 534, 536, 538, 540, 542, 544, 3291
messages 1640
native format, converting
to/from 528, 530, 532
ports
reading from 1075, 1079
writing to 1969, 1973
unaligned values
accessing safely 3603, 3605, 3607
writing safely 3597, 3599, 3601

lltoa() 1573
LOCAL_CREDNS 3617

local network addresses, converting
to/from IP addresses 1091
local times, converting to/from
calendar times 1539, 1541, 1668
LOCALDOMAIN 2515, 2519, 2523, 2525, 2528, 2531
localeconv() 1534
locales
classes, wide-character 3772
daylight saving time 415
numeric formatting 1534
setting 2880
strings, comparing 3219, 3721

localtime_r() 1541
localtime() 1539
LOCK_EX 698
LOCK_NB 698
LOCK_SH 698
LOCK_UN 698

lockf() 1544
locks
files 653, 698, 701, 792, 1544
mutexes
attributes, destroying 2272
attributes, initializing 2286
attributes, priority
ceiling 2274, 2289
attributes,
process-shared 2279, 2293
attributes, recursive 2281, 2295
attributes, scheduling
protocol 2276, 2292
attributes, type 2283, 2298
Index

destroying 2253, 3312
events 3310, 3314
initializing 2257, 3331
locking 2259, 2265, 2268, 3317, 3322
priority 3310
priority ceiling 2255, 2263
reviving 3320
unlocking 2270
read-write
attributes, creating 2328
attributes, destroying 2324
attributes, process-shared 2326, 2330
destroying 2303, 3312
initializing 2305, 3331
locking for reading 2308, 2310, 2316
locking for writing 2313, 2318, 2322
unlocking 2320
sleepon
destroying 3048
initializing 3050
locking 2349, 3052
unblocking 2347, 2351, 3046, 3054
unlocking 2357, 3056
waiting 2353, 2359, 3058
log
bankruptcy (not implemented) 2719
LOG_* (logging facilities) 1961
LOG_ALERT 3346
LOG_CRIT 3346
LOG_DEBUG 3346
LOG_EMERG 3346
LOG_ERR 3346
LOG_INFO 3346
LOG_MASK() 2882
LOG_NOTICE 3346
LOG_UPTO() 2882
LOG_WARNING 3346
log(), logf() 1547
log, system message
  closing 366
  log priority mask 2882
  opening 1962
  writing to 3345, 3687
    blocks 3060
    formatted output 3063, 3673
    integers 3066
log10(), log10f() 1551
log1p(), log1pf() 1549
logarithms
  base 10 1551
  integral 1069
  natural 1547
  \(x + 1\) 1549
logb(), logbf() 1553
logging in
  previous lines, discarding 1697
    pseudo-ttys 723, 1556
logical interrupt vector
  numbers 1155
LOGIN_PROCESS 995, 998, 3632
login_tty() 1556
long integers
  absolute value 1506
  division 1515
  pseudo-random numbers
    nonnegative 1561, 1936
    signed 1499, 1757, 2430
Index

quotient 1515
remainder 1515
size of 113
longjmp() 1558, 2423
lowercase
characters, converting to 3528
strings, converting to 3239
testing a character for 1451, 1476
wide characters, converting to 3532, 3534, 3770
lrand48() 1561
lsearch() 1563
lseek(), lseek64() 1567
resource managers,
implementing in 1283, 1286
lstat(), lstat64() 1570
ltoa() 1573
ltrunc() 1576

mallinfo() 1587
malloc() 1589
mallopt() 1591
manifests 113
MAP_ANON 1680
MAP_BELOW16M 1680
MAP_FAILED 1682, 1689
MAP_FIXED 1679, 1680, 1688, 1689
MAP_INLINE 1680
MAP_LOCAL 1680
MAP_NOX64K 1681
MAP_PHYS 1681
MAP_PRIVATE 1679
MAP_SHARED 1680
MAP_STACK 1681
MAP_TYPE 1679

mathematics
absolute values 121, 257, 638, 1506
Bessel functions 1493, 1495, 1497, 3802, 3804, 3806
complementary error function 563
division 465, 1515
error function 561
exponentials 630, 632, 764, 1513, 2665, 2668, 3012
finite numbers 693
floating-point settings 725, 728, 731, 734
gamma functions 813, 1520
hyperbolic functions 131, 166, 200, 400, 3042, 3364
hypotenuse, length of 1049
infinite numbers 1449
logarithms 1069, 1547, 1549, 1551

M
Machine Status Register 3350
main() 1584, 2983
arguments
 auxiliary 233
 number of 159
 parsing 906
 vector of 160
major device numbers 2558, 2648, 3195
major() 3195
makedev() 3195
mallinfo 1587

September 10, 2007
maximum 1595
minimum 1652
modular arithmetic 707
next representable
number 1927
not a number, determining
if 1453
powers 2092
pseudo-random numbers 488,
559, 1148, 1499, 1511,
1561, 1757, 1936, 2426,
2428, 2430, 2785, 2917,
3180, 3182, 3184
radix-independent
exponents 1553
remainders 490, 2507
roots 275, 3178
rounding 277, 703, 2612
sign, copying 396
trigonometry 129, 164, 196,
198, 398, 3040, 3362
max() 1595
MAXHOSTNAMELEN 2865
Maximum Segment Size
(MSS) 3391
MB_CUR_MAX 3767
mblen() 1597
mbrlen() 1600
mbtowc() 1602
mbsinit() 1605
mbsrtowcs() 1607
mbstate_t 1605
mbstowcs() 1609
mbtowc() 1612
MCHECK_DISABLED 1721
MCHECK_FREE 1721
MCHECK_HEAD 1721
MCHECK_OK 1721
mcheck_status 1721
MCHECK_TAIL 1721
mcheck() 1615
mem_offset(), mem_offset64()
1618
memalign() 1621
members, offset of within a
structure 1944
memccpy() 1623
memchr() 1625
memcmp() 1627
memcpy() 1629
memcpyv() 1631
memicmp() 1633
memmove() 1635
memory
allocating
__amblksiz 3068
aligned 1621, 2083
array 273, 2671
automatic (from stack) 152
blocks 1589, 2478, 3068
break value, changing 157,
2662
consistency check 1615,
1720
controlling 1591
data segment, changing 246
heap block, aligned on page
boundary 3651
information about,
getting 1587
comparing 1627, 1633, 3776
copying 1623, 1629, 1631,
3778
Index

overlapping objects 1635, 3780

devices
I/O, mapping 1685, 1873
physical, mapping into
process’s address space 1688
direct memory access (DMA)
  channels, managing 2633
free, amount of 3197
freeing 292, 754, 2478, 2929, 3186
locking 1674, 1867
  maximum amount 2903
  managing 2633
mapped
  contiguous length 1618
  maximum size 2903
  offset of 1618
physical storage, synchronizing
  with 1864
reallocating 2478, 3186
searching
  for a character 1625
  for a wide character 3774
setting 1637, 3782
shared
  mapping 1679
  unmapping 1871
unlocking 1869
unmapping 1875
memset() 1637
message_attach() 1640
message_attr_t 1640
message_connect() 1646
message_detach() 1649
message queues
attributes 1728, 1745
closing 1726
messages
  receiving 1739, 1749
  sending 1742, 1752
notifying when
  nonempty 1731
opening 1736
persistency of 1737
receive-only 1734
send-only 1734
send-receive 1734
unlinking 1754
messages
channels
  attaching to a process 380
  creating 300, 1880
  destroying 307, 1889
dispatch interface
  handlers 1640, 1649
errors, handling 1770
information about
  getting 1773
  structure 1759
Internet domain name servers
  errors 1020, 1026, 1035
  queries 2519, 2522, 2525, 2528, 2530
sending and
  interpreting 484, 486, 2515
key, adding 1776
reading data 1783, 1787, 2578, 2580
receiving 1791, 1805
replying 1809, 1812
resource managers
blocking while waiting for 2542
handling 2566
sending 1816, 1821, 1830, 1834, 1842, 1846, 1850
SNMP
  creating 3080
  freeing 3072
  reading 3082
  sending 3087
sockets
  peeking at 2483, 2486, 2490
  receiving from 2484, 2487, 2491
  sending to 2831, 2834, 2837
tampering, preventing 1776
unblocking 1770
writing data 1856, 1860, 2582, 2584
\textit{min}() 1652
minor device numbers 2558, 2648, 3195
\textit{minor}() 3195
misaligned access response 3447
\textit{mkdir}() 1654, 3593
\textit{mkfifo}() 1657, 3593
\textit{mknod}() 1220, 1660
  resource managers, implementing in 1289
\textit{mkstemp}() 1664
\textit{mktemp}() 1666
\textit{mktime}() 1668
\textit{mlockall}() 1674
\textit{mmap\_device\_io}() 1685
\textit{mmap\_device\_memory}() 1688
\textit{mmap}, \textit{mmap64}() 1679
resource managers,
  implementing in 1291, 1295
MODEM\_ALLOW8BIT 1696
MODEM\_ALLOWCASE 1696
MODEM\_ALLOWCTRL 1696
MODEM\_BAUD 1702, 1706
MODEM\_LASTLINE 1697
MODEM\_NOECHO 1702, 1703, 1706
\textit{modem\_open}() 1691
\textit{modem\_read}() 1697
\textit{modem\_script} 1701
\textit{modem\_script}() 1701
\textit{modem\_write}() 1708
modems
  opening 1691
  reading 1697
  script, running on 1701
  states 1703
  writing 1708
  escape characters 1708
  special characters 1708
\textit{modf}, \textit{modff}() 1711
modular arithmetic, floating point 707
\textit{mount\_parse\_generic\_args}() 1716
\textit{mount}() 1714
\textit{mprobe}() 1720
\textit{mprotect}() 1723
\textit{mq\_attr} 1728
\textit{mq\_close}() 1726
\textit{mq\_getattr}() 1728
\textit{mq\_notify}() 1731
\textit{mq\_open}() 1736
MQ\_PRIO\_MAX 1742
mq_receive() 1739
mq_send() 1742
mq_setattr() 1745
mq_timedreceive() 1749
mq_timedsend() 1752
mq_unlink() 1754
mrand48() 1757
MS_ASYNC 1863
MS_INVALIDATE 1863
MS_INVALIDATE_ICACHE 1863
MS_SYNC 1863
MSG_ABORT 2779
MSG_ADDR_OVER 2779
MSG_CTRUNC 2492
MSG_DONTROUTE 1646
MSG_EOF 2779
MSG_EOR 2492
MSG_FLAG_CROSS_ENDIAN 457
MSG_FLAG_SIDE_CHANNEL 1646
MSG_OOB 971, 2483, 2486, 2490, 2492, 2830, 2833, 2836
MSG_PEEK 2483, 2486, 2490
MSG_TRUNC 2492
MSG_UNORDERED 2778
MSG_WAITALL 2483, 2486, 2490
MsgDeliverEvent(),
  MsgDeliverEvent_r() 1396, 1762
MsgError(), MsgError_r() 1770
msghdr 2491
MsgInfo(), MsgInfo_r() 1773
MsgKeyData(), MsgKeyData_r() 1776
MsgRead(), MsgRead_r() 1783
MsgReadv(), MsgReadv_r() 1787
MsgReceive(), MsgReceive_r() 1791
MsgReceivePulse(),
  MsgReceivePulse_r() 1797
MsgReceivePulsev(),
  MsgReceivePulsev_r() 1801
MsgReceivev(), MsgReceivev_r() 1805
MsgReply(), MsgReply_r() 1809
MsgReplyv(), MsgReplyv_r() 1812
MsgSend(), MsgSend_r() 1816
MsgSendnc(), MsgSendnc_r() 1821
MsgSendPulse(),
  MsgSendPulse_r() 1826
MsgSendsv(), MsgSendsv_r() 1830
MsgSendsvnc(), MsgSendsvnc_r() 1834
MsgSendv(), MsgSendv_r() 1838
MsgSendvnc(), MsgSendvnc_r() 1842
MsgSendvs(), MsgSendvs_r() 1846
MsgSendvsnc(), MsgSendvsnc_r() 1850
MsgVerifyEvent(),
  MsgVerifyEvent_r() 1854
MsgWrite(), MsgWrite_r() 1856
MsgWritev(), MsgWritev_r() 1860
MSS (Maximum Segment Size) 3391
msync() 1864
multibyte characters
  bytes, counting 1597, 1600
  wide characters
    conversion object 1605
wide characters, converting

to/from 1602, 1607, 1609,
1612, 3713, 3741, 3757,
3767
munlock() 1867
munlockall() 1869
munmap_device_io() 1873
munmap_device_memory() 1875
munmap() 1871
mutexes

attributes

destroying 2272
initializing 2286
priority ceiling 2274, 2289
process-shared 2279, 2293
recursive 2281, 2295
scheduling protocol 2276,
2292
type 2283, 2298
destroying 2253, 3312
events, attaching 3310, 3314
initializing 2257, 3331
locking 2259, 2265, 2268,
3317, 3322
priority 3310
ceiling 2255, 2263
reviving 3320
unlocking 2270

name_close() 1887
name_detach() 1889
NAME_FLAG_ATTACH_GLOBAL
1880, 1891
NAME_FLAG_DETACH_SAVEDPP
1889
NAME_MAX 1892
name_open() 1891
name servers

ersors 1020, 1026, 1035
initializing 2515
names

compressing 484
expanding 486
queries 2519, 2522, 2525,
2528, 2530
names

domain 372, 834, 2845
host 881, 2865
peer 914
socket 241, 964, 2433, 2608
thread 2237, 2339
NAN (not-a-number) 1453
nanoseconds

busy-waiting for 1896, 1898,
1901, 1903, 1905
threads, suspending for 1894
timespec, converting to/from
1938, 3517

nansleep() 1894
nanospin_calibrate() 1898
nanospin_count() 1901
nanospin_ns_to_count() 1905
nanospin_ns() 1903
nanospin() 1896
nap() 1908
napms() 1909

NAME_ATTACH_FLAG_GLOBAL
1880
name_attach_t 1881
name_attach() 1880

N
natural logarithms  1547
nbacncnect_result()  1913
nbacncnect()  1910
ND_LOCAL_NODE  1915
ND_NODE_CMP()  1915
ND2S_DIR_HIDE  1919
ND2S_DIR_SHOW  1919
ND2S_DOMAIN_HIDE  1919
ND2S_DOMAIN_SHOW  1919
ND2S_LOCAL_STR  1920
ND2S_NAME_HIDE  1920
ND2S_NAME_SHOW  1920
ND2S_QOS_HIDE  1920
ND2S_QOS_SHOW  1920
ND2S_SEP_FORCE  1920
NDEBUG  168
NETDB_INTERNAL  871, 1020, 1027
netent  1917
netmgr_ndtostr()  1918
netmgr_remote_nd()  1923
netmgr_strtond()  1925
network
database
closing  546
entries, getting  900, 902, 904
opening  2884
structure  1917
host entries
errors  1020, 1026, 1035
getting  864, 868, 870, 874, 876, 879
network interface addresses
freeing  758
getting  883
structure  1067
network numbers
IP addresses, converting
to/from  1091, 1096
strings, converting
to/from  1100
network-byte order
host-byte order, converting
to/from  1037, 1039, 1940, 1942
Neutrino classification  108
Neutrino kernel functions
TraceEvent()  3547
NEW_TIME  995, 3632
nextafter(), nextafterf()  1927
nftw(), nftw64()  1930
NGROUPS_MAX  384, 862, 2861
nice()  1934
NO_DATA  1021, 1027
NO_RECOVERY  1021, 1027
node descriptors
comparing  1915
current  3609
relative to a remote node  1923
strings, converting
to/from  1918, 1925
nodenames, translating addresses
into  895
NOFD  1618, 1679, 1680
NOFLSH  3415
nonlocal jumps  1558, 2872, 2979, 3025
normalized fractions  764
not a number, determining if  1453
nrand48()  1936
nsec2timespec()  1938
NSIG  2970
ntohl()  1940
ntohs() 1942
  numbers
    determining if
      finite 693
      infinite 1449
      not a number 1453
  formatting 1534
  maximum 1595
  minimum 1652
  next representable
    floating-point 1927
  strings, converting
    to/from 205, 207, 209,
    211, 1490, 1573, 3271,
    3274, 3281, 3284, 3589,
    3636
  wide-character strings,
    converting to/from 3747,
    3751, 3755, 3761

O

O_APPEND 653, 1200, 1332, 1947,
  1957, 3110, 3115, 3789
O_CLOEXEC 1947
O_CREAT 402, 1734, 1947, 2943,
  3110, 3111
O_DSYNC 1201, 1363, 1368, 1948,
  3795
O_EXCL 1201, 1734, 1948, 2004,
  2943, 3110
O_LARGEFILE 1201, 1948
O_NOCTTY 1201, 1948
O_NONBLOCK 653, 1343, 1354,
  1390, 1698, 1734, 1948,
open(), open64() 402, 1947, 3593
resource managers,
  implementing in 1327
opendir() 1954
openfil() 1957
  resource managers,
  implementing in 1327, 1330, 1334
openlog() 1962
openpty() 1964
operating system
  name 372, 3609
  release level 372
  target 113
  version 113, 372, 3609
OPOST 3414
optarg 906
opterr 907
optimization, compiling with 113
optind 906
options
  command-line
    parsing 906, 984
  mount, parsing 1716
  socket-level 967, 2914
optopt 907
other scheduling 2161
out-of-band (OOB) data
  determining if at mark 3099
  sending/receiving 2483, 2486, 2490
out16() 1969
out16s() 1971
out32() 1973
out32s() 1975
out8() 1965
out8s() 1967
outbe16() 1969
outbe32() 1973
outle16() 1969
outle32() 1973
output, formatted 739, 804, 2096, 3657, 3663, 3667, 3691, 3787
overlapping memory,
  copying 1635, 3780
ownership, changing of a file 316, 645, 1508

P

P_ALL 3705
P_NOWAIT 3129, 3134, 3139, 3144, 3156, 3161, 3166, 3170
P_NOWAITO 3129, 3134, 3139, 3144, 3156, 3161, 3166, 3170
P_OVERLAY 50, 3129, 3134, 3139, 3144, 3156, 3161, 3166, 3170
P_PGID 3705
P_PID 3705
P_WAIT 50, 3129, 3134, 3139, 3144, 3156, 3161, 3166, 3170
packets See also ROUTE
  routing 2621
  SNMP
    reading 3082
PAREN 3414
PARMRK 3413
Index

PARODD 3414
PARSTK 3414
partitions, adaptive  See adaptive partitions
passwd 932, 935, 938, 940, 943
passwords
database
  closing 548
  entries, getting for a user 935, 938, 940, 943
  entries, getting next 932
  rewinding 2895
  encrypting 406, 2414
  prompting for and reading 912
shadow database
  closing 551
  entry, reading 682, 977, 981
  entry, structure 2399
  entry, writing 2399
  rewinding 2916
PATH 58, 325, 597, 620, 2782, 3139, 3144, 3149, 3166, 3170
PATH_MAX 831, 1892
pathconf() 1980
  resource managers,
    implementing in 1336, 1339
pathfind(), pathfind_r() 1984
pathmgr_symlink() 1987
pathmgr_unlink() 1989
pathname delimiter in QNX
 Momentics documentation lv
paths
  base name 234
  directory name 438
names
  maximum length 737, 1981
patterns, matching 711, 1010, 1014
  truncating 737, 1981
  resolving 2481
resource managers
  attaching to 2535
  detaching from 2554
  getting 2593
pattern matching  See regular expressions
pause() 1991
PC Card server
  arming 1994
  attaching 1997
  card insertion/removal,
    notification of 1994
  CIS (Card Information Structure), reading 2007
  detaching 1999
  locking 2004
  socket setup information 2001
  unlocking 2008
pccard_arm() 1994
pccard_attach() 1997
pccard_detach() 1999
pccard_info() 2001
pccard_lock() 2004
pccard_raw_read() 2007
pccard_unlock() 2008
PCI
  addresses
    converting 2017
    testing 2017
  BIOS, determining if
    present 2038
classes, finding 2026
devices
  attaching 2012
configuration, reading 2040,
  2043, 2045, 2048
configuration, writing 2052,
  2055, 2058, 2061
detaching 2024
  finding 2026, 2029
  rescanning for 2050
functions, finding 2026
interrupts
  mapping 2035
  routing information 2032
memory, sharing 1682
server
  attaching 2010
  detaching 2022
pci_attach_device() 2012
pci_attach() 2010
PCI_BAD_REGISTER_NUMBER
  2041, 2044, 2046, 2049,
  2056, 2059, 2062
PCI_BUFFER_TOO_SMALL 2041,
  2044, 2046, 2056, 2059
pci_detach_device() 2024
pci_dev_info 2013
PCI_DEVICE_NOT_FOUND 2024,
  2027, 2030, 2041
pci_find_class() 2026
pci_find_device() 2029
PCI_INIT_ALL 2017
PCI_INIT_BASE0…
  PCI_INIT_BASE5 2017
PCI_INIT_IRQ 2017
PCI_INIT_ROM 2017
PCI_IO_ADDR() 2017
pci_irq_routing_options() 2032
PCI_IS_IO() 2017
PCI_IS_MEM() 2017
pci_map_irq() 2035
PCI_MASTER_ENABLE 2017
PCI_MEM_ADDR() 2018
PCI_PERSIST 2016, 2024
pci_present() 2038
pci_read_config() 2040
pci_read_config16() 2045
pci_read_config32() 2048
pci_read_config8() 2043
pci_rescan_bus() 2050
PCI_ROM_ADDR() 2018
PCI_SEARCH_BUSDEV 2017
PCI_SEARCH_CLASS 2016
PCI_SEARCH_VEND 2016
PCI_SEARCH_VENDEV 2016
PCI_SET_FAILED 2036
PCI_SHARE 2016
PCI_SUCCESS 2027, 2030, 2038,
  2044, 2046, 2049, 2056,
  2059, 2062
PCI_UNSUPPORTED_FUNCTION
  2036
pci_write_config() 2052
pci_write_config16() 2058
pci_write_config32() 2061
pci_write_config8() 2055
pclose() 2063, 2077
PDU (Protocol Data Unit)  See
  SNMP
peers, getting names of
  connected 914
Peripheral Component Interconnect
  See PCI
permissions
Index

changing 312, 642
files, on creation 3592
daemons 2106
`perror()` 2065
PF_INET 3339
PF_KEY 1406
PF_KEY_V2 1406
PF_ROUTE 1115
PIPE_BUF 3790
`pipe()` 2067
pipes
  bytes, writing atomically 737, 1981
closing 2063
creating 2067
opening 2076
reading from 2094, 2446, 2474
pointers, size of `void` 113
`poll()` 2069
`pollfd` 2069
POOL_FLAG_EXIT_SELF 3420
POOL_FLAG_USE_SELF 3420
`popen()` 2076
portable code 105
ports
  managing 2633
  privileged
    socket, binding to 244
    socket, getting for 2629
  reading from 1071, 1073, 1075, 1077, 1079, 1081
serial
  opening 1691
  reading 1697
  script, running on 1701
  writing 1708
  services, finding for 958
POSIX See also message queues;
  semaphores; threads
  signals 2988
  standards 105
  version supported 3335
`posix_memalign()` 2083
`pow()`, `powf()` 2092
PowerPC platforms, variable-length
  argument lists on 3640
powers 2092
`pread()`, `pread64()` 2094
precision, floating-point 731
printable, testing a character
  for 1447, 1455, 1474, 1478
`printf()` 2096
priorities
  adjusting 1934, 2688
  getting 924, 2685
  inheritance 301, 2259, 2265, 2270
  preventing 303
  maximum 2690
  minimum 2692
  setting 2704, 2708, 2891, 3123
process groups
  changing 2120
  creating 2886, 2911, 3124
  devices 3379, 3399
  ID, getting 916, 918
  joining 2886
  membership, inheriting 579, 585, 592, 603, 609, 615, 3120
  pulses, sending 1826
Index

remote node 3122
session of a controlling
terminal 3381
setting 2889, 3123
signals, sending 1501, 1504,
2957, 2993, 2996
SIGHUP 625
SIGURG 3103
status of 3701, 3708
waiting for 3705
processes See also threads
adaptive partition
getting 2744
setting 2726
address space
device I/O memory,
mapping 1688
limits 2903
locking 1674
unlocking 1869
alarms, scheduling 149, 3584
alignment 3122
analyzing 1180
arguments
auxiliary 233
maximum length 3334
number of 159
parsing 906
vector of 160
asynchronous message channels,
attaching to 177
background 413, 2106
termination, notification of 2108
buffers
allocating 191
child
closing file descriptors
in 3118, 3149
state change, waiting for 3695, 3699, 3702, 3706, 3709
zombie, preventing from becoming 2956, 2991, 3123
configurable limits 3334
connections
client, information about 384
detaching 181, 303, 388
flags, modifying 390
flush 185
information about 393
receiving 189
controlling terminal, path name 408
CPU time, maximum 2903
creating 577, 584, 591, 597, 601, 607, 613, 620, 719,
723, 3119, 3123, 3130, 3135, 3140, 3145, 3150,
3157, 3162, 3167, 3171
data segment, changing space allocated for 246
debugging 416, 3123
dynamically linked libraries
addresses, translating 468
closing 470
debugging 479
errors 472
opening 474
symbol, getting address of 481
environment 555
clearing 325
restoring 1558, 2979
saving 2872, 3025
environment variables
defining 2394, 2850
deleting 2394, 2850, 3622
getting 840
executable file
base name 2122
file descriptor 367
full path 368
executing 577, 584, 591, 597, 601, 607, 613, 620, 3123
execution time 351
execution time limit,
getting 2702
file-mode creation mask 3592
daemons 2106
files, maximum per 3335
forking 719, 723
group ID
effective 838, 2847, 2856, 2896
real 844, 2856, 2896
saved 2856
supplementary 862, 2861
guardian, specifying 2115
holding for debugging 3123
I/O privileges,
requesting 1155, 1164, 1169, 1172, 1174, 1182, 1184, 1187, 1189, 3449
ID, getting 920, 922
interrupts
disabling 1172
enabling 1174
events 1163, 1169
handlers 1155, 1169, 1182, 1187
handlers, idle 1176
masking 1184, 1189
waiting for 1191
maximum per real user
ID 2903, 3334
memory, sharing 1681
message channels, attaching to 380
name 368, 2122
parent
blocking 3655
ID, getting 922
priority
adjusting 1934, 2688
getting 924, 2685
inheritance 301, 2259, 2265, 2270
inheritance, preventing 303
maximum 2690
minimum 2692
setting 2704, 2708, 2891, 3123
processor affinity 3450
program entry function 1584
resident set size,
maximum 2903
scheduling parameters 3123
scheduling policy
getting 2694
setting 2708, 3123
sessions 962, 2889, 2911
set-group ID 3335
set-user ID 3335
SIGALRM sending to 3466, 3492, 3584
signals
  actions for 2954, 2983, 2988
  information about 3038
  pending 3017
  queuing 3022
  raising 2423
  sending 1501, 1504, 2996
  suspending until delivered 3006
  waiting for 1991, 3009, 3031, 3036, 3038
  spawning 3119, 3130, 3135, 3140, 3145, 3150, 3157, 3162, 3167, 3171
  spawning and blocking 3655
  stack size, setting maximum for 3124
  supplementary group IDs, maximum 3334
  suspending 421, 1894, 1908, 1909, 3044, 3624, 3695, 3699, 3702, 3706, 3709
  system commands, executing 3356
  system-wide events notification of 2109
  triggering 2113
  terminating 119, 624, 627
  diagnostics 168
  functions to be called, registering 202
  time
    clock ID 332
    clock ticks 330
    time-accounting information 3513
  user ID
  effective 842, 2853, 2899, 2921
  real 991, 2899, 2921
  saved 2921
  user name, getting 891, 893
  yielding 2710
  zombies, preventing children from becoming 2956, 2991, 3123
  processor affinity 3450
  Processor Version Register 3350
  PROCDBG_DAEMON_KEEPUMASK 2106
  PROCDBG_DAEMON_NOCHDIR 2106
  PROCDBG_DAEMON_NOCLOSE 2106
  PROCDBG_DAEMON_NODEVNULL 2106
  procmgr_daemon() 2106
  PROCDBG_EVENT_DAEMON_DEATH 2108
  procmgr_event_notify() 2109
  PROCDBG_EVENT_SYNC 2108
  procmgr_event_trigger() 2113
  procmgr_guardian() 2115
  PROCDBG_SESSION_SETPGRP 2120
  PROCDBG_SESSION_SETSID 2120
  PROCDBG_SESSION_SIGNAL_LEADER 2120
  PROCDBG_SESSION_SIGNAL_PGRP 2120
  PROCDBG_SESSION_SIGNAL_PID 2120
PROCMGR_SESSION_TCSETSID 2120
procmgr_session() 2120
program entry function 1584
PROT_EXEC 1293, 1678, 1687, 1723
PROT_NOCACHE 1293, 1678, 1687, 1723
PROT_NONE 1293, 1678, 1687, 1723
PROT_READ 1293, 1678, 1687, 1723
PROT_WRITE 1293, 1678, 1687, 1723
Protocol Data Unit (PDU) See SNMP
protocols
database
closing 547
entries, getting 926, 928, 930
entry structure 2123
opening 2893
ICMP (Internet Control Message Protocol) 1054
ICMP6 (Internet Control Message Protocol v6) 1056
INET6 (Internet protocol v6 family) 1114
interprocess
communication 3616
IP (Internet Protocol) 1400
IP6 (Internet Protocol v6) 1423
IPsec (secure Internet Protocol) 1406
TCP (Transmission Control Protocol) 3390
UDP (User Datagram Protocol) 3587
protoent 2123
proxy server (SOCKS) 3811
pseudo-random numbers
double 488, 559
int 2426, 2428
long
  nonnegative 1561, 1936
  signed 1499, 1757, 2430
seed, setting 2785, 3180, 3182, 3184
sequence, initializing 3182
state
  initializing 1148, 1511
  switching 2917
pseudo-ttys
opening 723, 1964
preparing for a login 723, 1556
pthread_abort() 2124
PTHREAD_ABORTED 2124
pthread_atfork() 2126
pthread_attr_destroy() 2128
pthread_attr_getdetachstate() 2130
pthread_attr_getguardsize() 2132
pthread_attr_getinheritsched() 2134
pthread_attr_getschedparam() 2136
pthread_attr_getschedpolicy() 2138
pthread_attr_getscope() 2140
pthread_attr_getstackaddr() 2142
pthread_attr_getstacklazy() 2144
Index

pthread_getcpuclockid() 2235
pthread_getname_np() 2237, 2339
pthread_getschedparam() 2239
pthread_getspecific() 2241
PTHREAD_INHERIT_SCHED 2150, 2157
pthread_join() 2243
pthread_key_create() 2245
pthread_key_delete() 2249
pthread_kill() 2251
PTHREAD_MULTISIG_ALLOW 2224, 3443
PTHREAD_MULTISIG_DISALLOW 2224, 3443
PTHREAD_MUTEX_DEFAULT 2298
pthread_mutex_destroy() 2253
PTHREAD_MUTEX_ERRORCHECK 2297
pthread_mutex_getprioceiling() 2255
pthread_mutex_init() 2257
PTHREAD_MUTEX_INITIALIZER 2257
pthread_mutex_lock() 2259
PTHREAD_MUTEX_NORMAL 2297
PTHREAD_MUTEX_RECURSIVE 2297
pthread_mutex_setprioceiling() 2263
pthread_mutex_timedlock() 2265
pthread_mutex_trylock() 2268
pthread_mutex_unlock() 2270
pthread_mutexattr_destroy() 2272
pthread_mutexattr_getprioceiling() 2274
pthread_mutexattr_getprotocol() 2276
pthread_mutexattr_getpshared() 2279
pthread_mutexattr_getrecursive() 2281
pthread_mutexattr_gettype() 2283
pthread_mutexattr_init() 2286
pthread_mutexattr_setprioceiling() 2289
pthread_mutexattr_setprotocol() 2292
pthread_mutexattr_setpshared() 2293
pthread_mutexattr_setrecursive() 2295
pthread_mutexattr_settype() 2298
PTHREAD_ONCE_INIT 2300
pthread_once() 2300
PTHREAD_PRIO_INHERIT 2286, 2291, 3330
PTHREAD_PRIO_NONE 2291
PTHREAD_PRIO_PROTECT 2291, 3330
PTHREAD_PROCESS_PRIVATE 2185, 2216, 2293, 2330, 2365
PTHREAD_PROCESS_SHARED 2185, 2216, 2220, 2293, 2330, 2365
PTHREAD_RECURSIVE_DISABLE 2281, 2286, 2295
PTHREAD_RECURSIVE_ENABLE 2281, 2295
PTHREAD_RMUTEX_INITIALIZER 2257
pthread_rwlock_destroy() 2303
pthread_rwlock_init() 2305
PTHREAD_RWLOCK_INITIALIZER 2305
pthread_rwlock_rdlock() 2308
pthread_rwlock_timedrdlock() 2310
pthread_rwlock_timedwrlock() 2313
pthread_rwlock_tryrdlock() 2316
pthread_rwlock_trywrlock() 2318
pthread_rwlock_unlock() 2320
pthread_rwlockattr_destroy() 2324
pthread_rwlockattr_getpshared() 2326
pthread_rwlockattr_init() 2328
pthread_rwlockattr_setpshared() 2330
PTHREAD_SCOPE_PROCESS 3443
PTHREAD_SCOPE_SYSTEM 2150, 2163, 3443
pthread_self() 2332
pthread_setcancelstate() 2333
pthread_setcanceltype() 2335
pthread_setconcurrency() 2337
pthread_setschedparam() 2342
pthread_setspecific() 2343
pthreadSigmask() 2345
pthread_sleepon_broadcast() 2347
pthread_sleepon_lock() 2349
pthread_sleepon_signal() 2351
pthread_sleepon_timedwait() 2353
pthread_sleepon_unlock() 2357
pthread_sleepon_wait() 2359
pthread_spin_destroy() 2363
pthread_spin_init() 2365
pthread_spin_lock() 2367
pthread_spin_trylock() 2369
pthread_spin_unlock() 2371
PTHREAD_STACK_LAZY 2167
PTHREAD_STACK_MIN 2165, 2166, 2171, 2172, 3442, 3446
PTHREAD_STACK_NOTLAZY 2167
pthread_testcancel() 2373
pthread_timedjoin() 2374
pulse_attach() 2380
pulse_detach() 2383
pulses
  adaptive partition
  getting 2744
  setting 2727
  compression 1826
  dispatch interface
  attaching 2380
  detaching 2383
  priority of a receiving
  thread 2969
  queueing 1826
  receiving 1797, 1801
  sending 1826
  structure 2377
punctuation, testing a character
  for 1457, 1480
putc_unlocked() 2388
putc() 2386
putchar_unlocked() 2392
putchar() 2390
putenv()  598, 621, 2394, 3130, 3140, 3157, 3167
puts()  2397
putspent()  2399
pututline()  2402
putw()  2405
putwc()  2407, 2409
pwrite()  2411
random numbers
  double  488, 559
  int  2426, 2428
  long
    nonnegative  1561, 1936
    signed  1499, 1757, 2430
  seed, setting  2785, 3180, 3182, 3184
  sequence, initializing  3182
  state
    initializing  1148, 1511
    switching  2917
random()  2430
raw input mode
  buffer  736, 1980
  conditions for input
    request  2459
  FORWARD qualifier  2460
  MIN qualifier  2459
  TIME qualifier  2459
  TIMEOUT qualifier  2460
Rbind()  2433
rcmd()  2435, 3812
Rconnect()  2438
rcvid (receive identifier)  1792, 1806
  checking validity of  1854
rdchk()  2440
re_comp()  2442
re_exec()  2444
read_main_config_file()  2451
read()  1222, 2446, 2458
  resource managers, implementing in  1341, 1344
read-write locks
  attributes
creating  2328
destroying  2324
process-shared  2326, 2330
destroying  2303, 3312
initializing  2305, 3331
locking
   for reading  2308, 2310, 2316
   for writing  2313, 2318, 2322
unlocking  2320

readblock()  2455
readcond()  2458
readdir_r()  2468
readdir()  1954, 2464
readlink()  2471
resource managers,
   implementing in  1347

readv()  2474
realloc()  2478
realpath()  2481
realtime timers
   busy-waiting  1896, 1901, 1903, 1905
calibrating 1898
creating  3465
destroying  3469
expiry status  3471
overruns  3473
setting  3502
threads  340
time
   getting  3475
   setting  3478
rebooting  3348
rebooting on bankruptcy  2719
receive identifier  See rcvid
RECEIVED MESSAGE  3092
receiving
   messages  1791, 1805
   from a socket  2484, 2487, 2491
   pulses  1797, 1801
recv()  2484
recvfrom()  2487
recvmsg()  2491
REG_EXTENDED  2494
REG_ICASE  2494
REG_NOLINE  2494
REG_NOSUB  2494, 2503
REG_NOSTACK  2501
REG_NOTEOL  2501
regcomp()  2495
regerror()  2499
regex_t  2494
regexec()  2502
regfree()  2505
registers
   devices
      access to, gaining and
      relinquishing  1685, 1873
      reading  2040, 2043, 2045, 2048
      writing  2052, 2055, 2058, 2061
      floating-point exceptions  728
      Machine Status Register  3350
      Processor Version
         Register  3350
      real-mode software
         interrupts  1194
      TSC (Time Stamp Counter)  349
regmatch_t  2502
regular expressions
Index

basic 2495
compiling 2442, 2495
errors, explaining 2499
extended 2496
freeing 2505
string, comparing to 2444, 2502
remainder(), remainderf() 2507
remainders
  floating point 490, 2507
  integer 465
  long integer 1515
remote hosts
  commands, executing on 2435, 2627
  identity, checking 2657
remove() 2509
rename() 2512
RES_DEBUG 2515
RES_DEFNAMES 2516, 2528
RES_DNSRCH 2516, 2528
RES_INIT 2516
res_init() 2515
res_mkquery() 2519
res_query() 2522
res_querydomain() 2525
RES_RECURSE 2516
res_search() 2528
res_send() 2530
RES_STAYOPEN 2516
RES_USEVC 2516
residue, floating point 707
resmgr_attach() 2535
resmgr_attr_t 2536
resmgr_block() 2542
resmgr_connect_funcs_t 2545
resmgr_context_alloc() 2547
resmgr_context_free() 2550
resmgr_context_t 2552
resmgr_detach() 2554
resmgr_devino() 2558
RESMGR_FLAG_AFTER 1713, 1716
RESMGR_FLAG_ATTACH_OTHERFUNC 2537
RESMGR_FLAG_BEFORE 1713, 1716
RESMGR_FLAG_OPAQUE 1714, 1716
resmgr_handle_tune() 2563
resmgr_handler() 2566
resmgr_io_funcs_t 2572
resmgr_iofuncs() 2576
resmgr_msgread() 2578
resmgr_msgreadv() 2580
resmgr_msgwrite() 2582
resmgr_msgwritev() 2584
resmgr_open_bind() 2590
resmgr.pathname() 2593
resmgr_unbind() 2600
resolv.conf, contents of 372
resolver routines
  errors 1020, 1026, 1035
  initializing 2515
  Internet domain names
    compressing 484
    expanding 486
  options 2515
  queries 2519, 2522, 2525, 2528, 2530
resource database manager
  about 2633
  device numbers
attaching 2648
detaching 2652
resources
creating 2640
destroying 2644
querying 2655
reserving 2633
returning 2646
resource managers
access, checking 1229
arming for notification 1394
attaching 2535
attributes
initializing 1214
locking 1216, 1220, 1224, 1281
structure 1219
time members,
  updating 1370
unlocking 1226
clients
information about 1242
unblocking 1372, 1374
connect functions 2545
default values, setting 1265
open 1327
connect messages
file type reply 1205
link reply 1207
structure 1198
connection IDs 381
context
allocating 2547
freeing 2550
structure 2552
database, expanding
capacity 2561
detaching 2554
device number, getting 2558
device-control commands 425
file descriptors, mapping to
  OCBs 2563
function tables,
  initializing 1265
helper functions
chmod 1232
chown 1237
close 1244
devctl 1253
fdinfo 1260
link 1268
lseek 1283
mknod 1289
mmap 1291
open 1322
openfd 1330
pathconf 1336
read 1344
readlink 1347
rename 1351
space 1355
stat 1358
sync 1367
unlink 1377
utime 1382
write 1391
I/O functions
chmod 1235
chown 1240
client connection, getting
  for 2576
close 1247
default values, setting 1265
devctl 1257
| fdinfo     | 1261       | attaching | 1310, 2590       |
| lock       | 1276       | detaching | 1249, 1314, 2600 |
| lseek      | 1286       | file descriptors, mapping | to 2563 |
| mmap       | 1295       | freeing   | 1317            |
| OCB, close | 1251       | getting    | 2588            |
| OCB, lock  | 1281       | structure  | 1320            |
| OCB, unlock| 1380       | path       |                 |
| openfd     | 1334       | attaching to | 2535          |
| pathconf   | 1339       | detaching from | 2554          |
| read       | 1341       | getting    | 2593            |
| stat       | 1360       | server attributes, getting | 1212 |
| structure  | 2572       | status, returning | 2598 |
| sync       | 1365       | synchronization, checking to see | if required | 1363 |
| unblock    | 1374       | threads in | 1220            |
| utime      | 1385       | resources, system |       |
| write      | 1388       | creating   | 2640            |
| inode number, getting | 2558   | destroying | 2644            |
| iov_t      |           | limits     |                 |
| filling    | 2596       | getting    | 836, 946        |
| getting    | 2586       | setting    | 2902            |
| locks (not implemented) | 1272, 1274, 1279 | querying | 2655            |
| messages   |           | reserving  | 2633            |
| blocking while waiting | for 2542 | returning | 2646            |
| handling   | 2566       | usage, getting | 949            |
| reading    | 2578, 2580 | rewind()   | 2602            |
| writing    | 2582, 2584 | rewinded() | 2605            |
| notification |         | rftp       | 3811            |
| arming for | 1394       | Rgetsockname() | 2608          |
| installing, polling, and removing | 1299 | rindex()   | 2610            |
| removing for a client | 1304 | rint(), rintf() | 2612 |
| triggering | 1306       | RLIM_INFINITY | 2903, 2907     |
| Open Control Block (OCB) |          | RLIM_SAVED_CUR | 2907          |
| allocating | 1311       | RLIM_SAVED_MAX | 2907          |
|            |            | RLIMIT_AS    | 2903            |
|            |            | RLIMIT_CORE  | 2903            |
Index

© 2007, QNX Software Systems GmbH & Co. KG.

RTF_DYNAMIC 2624
RTF_GATEWAY 2624
RTF_HOST 2624
RTF_LLINFO 2624
RTF_MASK 2624
RTF_MODIFIED 2624
RTF_PROTO1 2624
RTF_PROTO2 2624
RTF_REJECT 2624
RTF_STATIC 2624
RTF_UP 2624
RTF_XRESOLVE 2624
RTLD_DEFAULT 481
RTLD_GLOBAL 477
RTLD_GROUP 478
RTLD_LAZY 477
RTLD_LOCAL 477
RTLD_NOW 477
RTLD_WORLD 478
RTM_ADD 2622
RTM_CHANGE 2622
RTM_DELADDR 2622, 2623
RTM_DELETE 2622
RTM_GET 2622
RTM_IFINFO 2622, 2623
RTM_LOSING 2622
RTM_MISS 2622
RTM_NEWADDR 2622, 2623
RTM_REDIRECT 2622
RTM_RESOLVE 2622
RTV_EXPIRE 2624
RTV_HOPCOUNT 2624
RTV_MTU 2624
RTV_RPIPE 2624
RTV_RTT 2624
RTV_RTTVAR 2624
RTV_SPIPE 2624

RTV_SSTHRESH 2624
RUN_LVL 995, 3632
rusage 949
ruserok() 2657

S

S_IEXEC 1202, 3193
S_IFBLK 1202, 1208, 3194
S_IFCHR 1202, 1208, 3194
S_IFDIR 1202, 1208, 1660, 3194
S_IFIFO 1202, 1208, 1660, 3194
S_IFLNK 1202, 1208, 3194
S_IFMT 1202, 1208, 3194
example of use 1445
S_IFNAM 1202, 1208, 3194
S_IFREG 1202, 1208, 3194
S_IFSOCK 1202, 1208, 1445, 3194
S_IREAD 1202, 3193
S_IRGRP 3192, 3592
S_IROTH 3192, 3592
S_IRUSR 3192, 3592
S_IRWXG 2814, 3192, 3592
S_IRWXO 2814, 3192, 3592
S_IRWXU 2814, 3192, 3592
S_ISBLK() 773, 3194
S_ISCHR() 773, 3194
S_ISDIR() 773, 3194
S_ISIFO() 773, 3194
S_ISGID 312, 313, 317, 642, 645, 774, 1509, 3193, 3573
S_ISLNK() 774, 3194
S_ISNAM() 3194
S_ISSOCK() 774, 3194
S_ISSOCK() 3195

3974  Index  September 10, 2007
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_ISUID</td>
<td>312, 313, 317, 642, 645, 774, 1509, 3193, 3573</td>
</tr>
<tr>
<td>S_ISVTX</td>
<td>312</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>3192, 3592</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>3192, 3592</td>
</tr>
<tr>
<td>S_IWRITE</td>
<td>1202, 3193</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>3192, 3592</td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>3192, 3592</td>
</tr>
<tr>
<td>S_IXOTH</td>
<td>3192, 3592</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>3192, 3592</td>
</tr>
<tr>
<td>S_TYPEISMQ()</td>
<td>774, 3195</td>
</tr>
<tr>
<td>S_TYPEISSEM()</td>
<td>774, 3195</td>
</tr>
<tr>
<td>S_TYPEISSHM()</td>
<td>774, 3195</td>
</tr>
<tr>
<td>SA_NOCLDSTOP</td>
<td>2984, 2990</td>
</tr>
<tr>
<td>SA_ONSTACK</td>
<td>578</td>
</tr>
<tr>
<td>SA_SIGINFO</td>
<td>1528, 2955, 2991, 3022</td>
</tr>
<tr>
<td>SAT (System Analysis Toolkit)</td>
<td>1180</td>
</tr>
<tr>
<td>sbrk()</td>
<td>2662</td>
</tr>
<tr>
<td>scalb(), scalbf()</td>
<td>2665</td>
</tr>
<tr>
<td>scalbn(), scalbnf()</td>
<td>2668</td>
</tr>
<tr>
<td>scandir()</td>
<td>2674</td>
</tr>
<tr>
<td>scanf()</td>
<td>2676</td>
</tr>
<tr>
<td>SCHED_ADJTOHEAD</td>
<td>2748</td>
</tr>
<tr>
<td>SCHED_ADJTOTAIL</td>
<td>2748</td>
</tr>
<tr>
<td>SCHED_APS_ADD_SECURITY</td>
<td>2740</td>
</tr>
<tr>
<td>SCHED_APS_ATTACH_EVENTS</td>
<td>2737</td>
</tr>
<tr>
<td>SCHED_APS_BNKR_BASIC</td>
<td>2718</td>
</tr>
<tr>
<td>SCHED_APS_BNKR_CANCEL_BUDGET</td>
<td>2718</td>
</tr>
<tr>
<td>SCHED_APS_BNKR_LOG</td>
<td>2719</td>
</tr>
<tr>
<td>SCHED_APS_BNKR_LOG (not implemented)</td>
<td>2719</td>
</tr>
<tr>
<td>SCHED_APS_BNKR_REBOOT</td>
<td>2719</td>
</tr>
<tr>
<td>SCHED_APS_BNKR_RECOMMENDED</td>
<td>2719</td>
</tr>
<tr>
<td>sched_aps_clear_crit parms</td>
<td>2735</td>
</tr>
<tr>
<td>SCHED_APS_CLEAR_CRITICAL</td>
<td>2735</td>
</tr>
<tr>
<td>sched_aps_create_parms</td>
<td>2721</td>
</tr>
<tr>
<td>SCHED_APS_CREATE_PARTITION</td>
<td>2721</td>
</tr>
<tr>
<td>sched_aps_events_parm</td>
<td>2738</td>
</tr>
<tr>
<td>sched_aps_info</td>
<td>2714</td>
</tr>
<tr>
<td>sched_aps_join_parms</td>
<td>2726</td>
</tr>
<tr>
<td>SCHED_APS_JOIN_PARTITION</td>
<td>2726</td>
</tr>
<tr>
<td>SCHED_APS_LOOK_UP</td>
<td>2725</td>
</tr>
<tr>
<td>sched_aps_lookup_parms</td>
<td>2725</td>
</tr>
<tr>
<td>sched_aps_mark_crit_parms</td>
<td>2734</td>
</tr>
<tr>
<td>SCHED_APS_MARK_CRITICAL</td>
<td>2734</td>
</tr>
<tr>
<td>sched_aps_modify_parms</td>
<td>2728</td>
</tr>
<tr>
<td>SCHED_APS_MODIFY_PARTITION</td>
<td>2728</td>
</tr>
<tr>
<td>sched_aps_overall_stats</td>
<td>2732</td>
</tr>
<tr>
<td>SCHED_APS_OVERALL_STATS</td>
<td>2732</td>
</tr>
<tr>
<td>sched_aps_parms</td>
<td>2720</td>
</tr>
<tr>
<td>sched_aps_partition_info</td>
<td>2723</td>
</tr>
<tr>
<td>sched_aps_partition_stats</td>
<td>2730</td>
</tr>
<tr>
<td>SCHED_APS_PARTITION_STATS</td>
<td>2730</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SCHED_APS_PSTATS_IS_BANKRUPT_NOW</td>
<td>2731</td>
</tr>
<tr>
<td>SCHED_APS_PSTATS_WAS_BANKRUPT</td>
<td>2731</td>
</tr>
<tr>
<td>SCHED_APS_QUERY_PARMS</td>
<td>2714</td>
</tr>
<tr>
<td>SCHED_APS_QUERY_PARTITION</td>
<td>2723</td>
</tr>
<tr>
<td>SCHED_APS_QUERY_PROCESS</td>
<td>2744</td>
</tr>
<tr>
<td>sched_aps_query_processParms</td>
<td>2744</td>
</tr>
<tr>
<td>SCHED_APS_QUERY_THREAD</td>
<td>2736</td>
</tr>
<tr>
<td>sched_aps_query_threadParms</td>
<td>2736</td>
</tr>
<tr>
<td>SCHED_APS_SCHEDPOL_BMP_SAFETY</td>
<td>2717</td>
</tr>
<tr>
<td>SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO</td>
<td>2716</td>
</tr>
<tr>
<td>SCHED_APS_SEC_BASIC</td>
<td>2741</td>
</tr>
<tr>
<td>SCHED_APS_SEC_FLEXIBLE</td>
<td>2741</td>
</tr>
<tr>
<td>SCHED_APS_SEC_JOIN_SELF_ONLY</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_LOCK_PARTITIONS</td>
<td>2740</td>
</tr>
<tr>
<td>SCHED_APS_SEC_NONZERO_BUDGETS</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_PARENT_JOINS</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_PARENT_MODIFIES</td>
<td>2742</td>
</tr>
<tr>
<td>SCHED_APS_SEC_PARTITION_LOCKED</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_RECOMMENDED</td>
<td>2741</td>
</tr>
<tr>
<td>SCHED_APS_SEC_ROOT_JOINS</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_ROOT MAKES_CRITICAL</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_ROOT MAKES_PARTITIONS</td>
<td>2742</td>
</tr>
<tr>
<td>SCHED_APS_SEC_ROOT0 OVERALL</td>
<td>2742</td>
</tr>
<tr>
<td>SCHED_APS_SEC_SYS_JOINS</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_SYS MAKES_CRITICAL</td>
<td>2743</td>
</tr>
<tr>
<td>SCHED_APS_SEC_SYS MAKES_PARTITIONS</td>
<td>2742</td>
</tr>
<tr>
<td>sched_aps_security_parms</td>
<td>2740</td>
</tr>
<tr>
<td>SCHED_APS_SET_PARMS</td>
<td>2720</td>
</tr>
<tr>
<td>SCHED_FIFO</td>
<td>2161, 2690, 2692, 2707, 2748, 2750, 3125</td>
</tr>
<tr>
<td>sched_get_priority_adjust()</td>
<td>2688</td>
</tr>
<tr>
<td>sched_get_priority_max()</td>
<td>2690</td>
</tr>
<tr>
<td>sched_get_priority_min()</td>
<td>2692</td>
</tr>
<tr>
<td>sched_getparam()</td>
<td>2685</td>
</tr>
<tr>
<td>sched_getscheduler()</td>
<td>2694</td>
</tr>
<tr>
<td>SCHED_NOCHANGE</td>
<td>2161, 3442</td>
</tr>
<tr>
<td>SCHED_OTHER</td>
<td>2161, 2690, 2692, 2707, 2748, 2750, 3125</td>
</tr>
<tr>
<td>sched_param</td>
<td>2696</td>
</tr>
<tr>
<td>SCHED_RR</td>
<td>2161, 2690, 2692, 2707, 2748, 2750, 3125</td>
</tr>
<tr>
<td>sched_rr_get_interval()</td>
<td>2702</td>
</tr>
<tr>
<td>sched_setparam()</td>
<td>2704</td>
</tr>
<tr>
<td>sched_setscheduler()</td>
<td>2708</td>
</tr>
<tr>
<td>SCHED_SPORADIC</td>
<td>2161, 2748, 3125</td>
</tr>
<tr>
<td>sched_yield()</td>
<td>2710</td>
</tr>
<tr>
<td>SchedCtl(), SchedCtl_r()</td>
<td>2713</td>
</tr>
</tbody>
</table>
SchedGet(), SchedGet_r() 2747
SchedInfo(), SchedInfo_r() 2751
SchedSet(), SchedSet_r() 2754
scheduling
information, getting 2751
parameters 2696
threads, getting for 2136, 2159, 2239, 2747
threads, inheriting 2134, 2157, 3443
threads, setting for 2342, 2754, 3123
policy
don’t change 2161
FIFO 2161
other 2161
processes, getting for 2694
processes, setting for 2708, 3123
round-robin 2161
sporadic 2161
threads, getting for 2138, 2239, 2747
threads, inheriting 2134, 2157, 3443
threads, setting for 2161, 2342, 2754, 3442
yielding 2757
SchedYield(), SchedYield_r() 2757
SCM_RIGHTS 3617
scoped addresses 1115
scripts, running 597, 620, 3119, 3123, 3130, 3135, 3140, 3145, 3151, 3157, 3162, 3167, 3171
SCTP
associations
adding or removing
addresses 2761
branching into a separate
socket 2773
locally bound addresses 2766, 2769
messages
receiving 2775
sending 2778
multihomed endpoints,
connecting to 2764
peer addresses 2767, 2771
protocol 2759
SCTP (Stream Control Transmission
Protocol)
classification 108
SCTP_BINDX_ADD_ADDR 2761
SCTP_BINDX REM_ADDR 2761
sctp_bindx() 2761
sctp_connectx() 2764
sctp_freeladdr() 2766
sctp_freeppaddrs() 2767
sctp_getladdr() 2769
sctp_getpaddrs() 2771
sctp_peeloff() 2773
sctp_recvmsg() 2775, 2778
searchenv() 2782
secure Internet Protocol See IPsec
secure RPC domain 372
Security Policy Database
(SPD) 1410
security, setting 2740
seed48() 2785
SEEK CUR 655, 768, 1283, 1356, 1566, 1576
SEEK_END 655, 768, 1283, 1356, 1566, 1576
SEEK_SET 654, 655, 768, 1283, 1356, 1566, 1576
seekdir() 2787
segments
  data
    maximum size 2903
data, end of 516, 519
text
    beginning of 252
    end of 576
select_attach() 2796
select_attr_t 2795
select_detach() 2799
SELECT_FLAG_EXCEPTION 2796
SELECT_FLAG_READ 2796
SELECT_FLAG_REARM 2796
SELECT_FLAG_SRVEXCEPT 2797
SELECT_FLAG_WRITE 2796
select_query() 2803
select() 124, 2790, 3812
data from snmp_select_info() 3085
sem_close() 2805
sem_destroy() 2807
sem_getvalue() 2809
sem_init() 2811
sem_open() 2815
sem_post() 2819
sem_timedwait() 2821
sem_trywait() 2824
sem_unlink() 2826
SEM_VALUE_MAX 2811, 2812, 2815
sem_wait() 2828
semaphores
  named
accessing and creating 2815
closing 2805
destroying 2826
posting 2819, 3325
unnamed
  destroying 2807, 3312
  initializing 2811, 3331
value
  decrementing 2821, 2824, 2828, 3327
  getting 2809
  incrementing 2819, 3325
  setting 2811, 2815
  waiting on 2828, 3327
  with a time limit 2821
  without blocking 2824
send() 2831
sendmsg() 2834
sendto() 2837
serial number, getting 372
serial ports
  opening 1691
  reading 1697
  script, running on 1701
  writing 1708
servent 2839
server attributes, getting 1212
servers
  connections
    creating 172, 300
    destroying 174, 307
  information about 384, 393
data server
  applications, registering and
deregistering 497, 503
  variables, creating and
destroying 492, 495
variables, flags 499
variables, getting and setting 501, 505
PCI
attaching 2010
detaching 2022
services
database
closing 550
entries, getting 956, 958, 960
entry structure 2839
opening 2909
sessions
character device terminal
drivers, support for 2120
controlling terminal 3381
creating 1556, 2911
current 2591
disassociating 625
ID, getting 962
leader, creating 2889, 3124
membership, inheriting 579, 585, 592, 603, 609, 615, 3151
remote node 3122
system daemons 2106
termination, notification of 2108
SET_REQ_MSG 3080
setbuf() 2840
setbuffer() 2842
setdomainname() 2845
setegid() 2847
setenv() 598, 621, 2850, 3130, 3140, 3157, 3167
seteuid() 2853
setgid() 2856
setgrent() 2859
setgroups() 2861
sethostent() 865, 871, 2863
sethostname() 2865
SETIOV() 2867
setitimer() 2869
setjmp() 2872
setkey() 2875
setlinebuf() 2877
setlocale() 2880, 3288
setlogmask() 2882
setnetent() 2884
setpgid() 2886
setpgrp() 2889
setprio() 2891
setprotoent() 2893
setpwent() 2895
setregid() 2896
setreuid() 2899
setrlimit(), setrlimit64() 2902
setservent() 2909
setsid() 2911
setsockopt() 2914
setspent() 2916
setstate() 2917
settimeofday() 2919
setuid() 2921
not honoring on mounted filesystems 1713
setutent() 2924
setvbuf() 2926
sh 2076
SH_COMPAT 1203, 1332, 3111, 3115
SH_DENYNO 1203, 1332, 3111, 3115
SH_DENYRD  1203, 1332, 3111, 3115
SH_DENYRW  1203, 1332, 3111, 3115
SH_DENYWR  1203, 1332, 3111, 3115

shadow password database
  closing  551
  entries
  reading  682, 977, 981
  structure  2399
  writing  2399
  rewinding  2916

shared locks  699

shared memory
  access protection,
    changing  1723
  attributes, modifying  2932, 2941
  mapping  1679
  opening  2944
  removing  2950
  unmapping  1871

shared objects
  addresses, translating  468
  closing  470
  debugging  479
  errors  472
  opening  474
  symbol, getting address of  481

SHELL  325, 2076, 3357

shell scripts, running  597, 620, 3119, 3123, 3130, 3135, 3140, 3145, 3151, 3157, 3162, 3167, 3171

shm_ctl_special()  2941
shm_ctl()  2932

shm_open()  2944
shm_unlink()  2950
SHMCTL_ANON  2931, 2940
SHMCTL_GLOBAL  2931, 2940
SHMCTL_LAZYWRITE  2931, 2940
SHMCTL_LOWERPROT  2931, 2940
SHMCTL_PHYS  2931, 2940
SHMCTL_PRIV  2931, 2940
shutdown()  2952
SIASYNCIO  2957, 2992, 2998
SI_MAXAVAL  2971
SI_MESGQ  2957, 2992, 2998
SI_MINAVAL  2971
SI_QUEUE  2957, 2992, 2998
SI_TIMER  2957, 2992, 2998
SI_USER  2957, 2992, 2998
side channels  380
SIG_BLOCK  3019
SIG_DFL  578, 2956, 2983
SIG_ERR  2985
SIG_IGN  578, 2956, 2983, 2991, 3124
SIG_INTR  2718
SIG_SETMASK  3019
SIG_UNBLOCK  3019
SIGABRT  119, 2988

sigaction  2990
sigaction()  2954
sigaddset()  2960
SIGALRM  149, 890, 2870, 2989, 3489
  process, sending to  3466, 3492, 3584
sigblock()  2962
SIGBUS  1680, 2989
SIGCHLD 625, 949, 2955, 2984, 2985, 2989, 3706
default actions 2956, 2991
ignoring 2956, 2991
SIGCONT 625, 2962, 2989
default actions 2956
sigdelset() 2964
sigemptyset() 2966
SIGEV_CLEAR_CRITICAL() 2973
SIGEV_GET_TYPE() 2969
SIGEV_INTR 1158, 1164, 1191, 1763, 2968
SIGEV_INTR_INIT() 2969
SIGEV_MAKE_CRITICAL() 2973
SIGEV_NONE 1528, 2738, 2968
SIGEV_NONE_INIT() 2738, 2969
SIGEV_PULSE 1158, 1164, 1732, 1763, 2968, 3466
SIGEV_PULSE_INIT() 2970
SIGEV_PULSE_Prio_inherit 2969
SIGEV_SIGNAL 1159, 1164, 1528, 1732, 1763, 2968, 3466
SIGEV_SIGNAL_CODE 1159, 1164, 1732, 2968, 3466
SIGEV_SIGNAL_CODE_INIT() 2971
SIGEV_SIGNAL_INIT() 2970
SIGEV_SIGNAL_THREAD 1159, 1164, 1732, 2968, 3466
SIGEV_SIGNAL_THREAD_INIT() 2971
SIGEV_SIGNAL_VALUE_INIT() 2970
SIGEV_THREAD 2968
SIGEV_THREAD_INIT() 2972
SIGEV_UNBLOCK 1763, 2968
SIGEV_UNBLOCK_INIT() 2972
sigevent 1396, 1762, 2968
business 2718, 2737
critical 2718, 2734
system overload 2737
type, determining
withSIGEV_GET_TYPE() 2969
sigfillset() 2975
SIGFPE 2989
SIGHOLD 3123
SIGHUP 625, 2988
process groups, targeting 625
SIGILL 2988
siginfo_t 2956, 2992, 2998
SIGINT 2988
process groups, sending
to 3399
SIGIO 2989
default actions 2956, 2991
SIGIOT 2988
sigismember() 2977
SIGKILL 2345, 2957, 2962, 2984, 2985, 2989
siglongjmp() 2979
sigmask() 2981
sign, copying 396
signal() 2983
SignalAction(), SignalAction_r() 2988
SignalKill(), SignalKill_r() 2996
SignalProcmask(),
SignalProcmask_r() 3002
signals
actions 2954, 2983, 2988
  default 2955
blocking 2962, 3027
  SIGCONT 2962
  SIGKILL 2345, 2962
  SIGSTOP 2345, 2962
  SIGTTTOU 748, 750, 2408, 2410
catching
  SIGKILL 2984
  SIGSTOP 2984
ignoring 2956
  SIGCHLD 949, 2985
  SIGKILL 2957, 2985
  SIGSTOP 2957, 2985
  SIGTTTOU 748, 750, 2408, 2410
information about 3038
masks
  constructing 2981
  restoring 2979
  saving 3025
  signal-blocked 3002
  threads 2345
names 2988
POSIX 2988
process groups, targeting 625, 1501, 1504, 2957, 2993, 2996
processes
  pending 3017
  queuing 2955, 3022
  suspending until delivered 3006
  sending 1501, 1504, 2423, 2996
SIGABRT 119
SIGALRM 149, 890, 2870
SIGBUS 1680
SIGCHLD 625, 2955, 2984
SIGCONT 625
SIGHOLD 3123
SIGHUP 625
SIGKILL 2984
SIGPIPE 748, 750, 970, 1707, 1709, 2408, 2410, 2412
SIGSEGV 1164, 1871, 1875, 2155, 2903
SIGTRAP 416
SIGXCPU 2903
SIGXFSZ 2903
sets
  adding to 2960
  initializing 2966, 2975
  membership, checking for 2977
  removing from 2964
string describing 3265
threads
  mask 3019, 3029
  threads, targeting 2251, 2957, 2993, 2996
  unblocking 3034
  user-defined 2989
  waiting for 1991, 3009, 3015, 3029, 3031, 3036, 3038
  SignalSuspend(),
    SignalSuspend_r() 3006
  SignalWaitinfo(),
    SignalWaitinfo_r() 3009
  significand(), significandf() 3012
  sigpause() 3015
  sigpending() 3017
SIGPIPE  748, 750, 970, 1707, 1709, 2408, 2410, 2412, 2989
SIGPOLL  2989
sigprocmask()  3019
SIGPWR  2989
sigqueue()  3022
SIGQUIT  2988
SIGRTMAX  2990
SIGRTMIN  2990
SIGSEGV  1164, 1187, 1189, 1871, 1875, 2155, 2903, 2989
sigsetjmp()  3025
sigsetmask()  3027
SIGSTOP  2345, 2957, 2962, 2984, 2985, 2989
  default actions  2956
sigsuspend()  3029
SIGSYS  2989
SIGTERM  2989
sigtimedwait()  3031
SIGTRAP  416, 2988
SIGTSTP  2989
SIGTIN  2989, 3656
SIGTTOU  748, 750, 2408, 2410, 2989, 3656
sigunblock()  3034
SIGURG  2989
  default actions  2956, 2991
  process groups, sending
to  3103
SIGUSR1  2989
SIGUSR2  2989
sigwait()  3036
sigwaitinfo()  3038
SIGWINCH  2989
  default actions  2956, 2991
SIGXCPU  2903
SIGXFSZ  2903, 3573
Simple Network Management
Protocol  See SNMP
sin(), sinf()  3040
sines  3040
  hyperbolic  3042
  inverse hyperbolic  166
sinh(), sinh()  3042
SIOC GIFCONF  1403
SIOC GIFFLAGS  1403
site-local addresses  1115
sleep()  3044
sleeping
  for microseconds  3624
  for milliseconds  421, 1908, 1909
sleep on locks
  destroying  3048
  initializing  3050
  locking  2349, 3052
  unblocking  2347, 2351, 3046, 3054
  unlocking  2357, 3056
  waiting  2353, 2359, 3058
slogb()  3060
slogf()  3063
slogi()  3066
SNMP (Simple Network
Management Protocol)
classification  108
daemon, configuration file
  for  2451
messages
  creating  3080
  freeing  3072
  reading  3082
  sending  3087
Protocol Data Unit (PDU)
  creating 3080
  freeing 3072
  processing 3082
  sending 3087
  structure 3076
sessions
  characteristics 3090
  closing 3070
  opening 3074
  timeouts, handling 3094
transactions,
  asynchronous 3085
snmp_close() 3070
SNMP_DEFAULT_ADDRESS 3091
SNMP_DEFAULT_COMMUNITY_LEN 3090
SNMP_DEFAULT_ENTERPRISE_LENGTH 3077
SNMP_DEFAULT_ERRINDEX 3077
SNMP_DEFAULT_ERRSTAT 3077
SNMP_DEFAULT_PEERNAME 3091
SNMP_DEFAULT_REMPORT 3091
SNMP_DEFAULT_REQID 3077
SNMP_DEFAULT_RETRIES 3090
SNMP_DEFAULT_TIME 3078
SNMP_DEFAULT_TIMEOUT 3090
snmp_errno 3070, 3074, 3081, 3088
snmp_free_pdu() 3072
snmp_open() 3074
snmp_pdu 3076
snmp_pdu_create() 3080
snmp_read() 3082
  using with select() 3085
snmp_select_info() 3085
  using with select() 3085
snmp_send() 3087
snmp_session 3074, 3090
snmp_timeout() 3094
  using with select() 3085
SNMP_VERSION_1 3076, 3092
SNMP_VERSION_2 3076, 3092
SNMPCONFIGFILE 2453
snmpd_conf_data 2451
snmpd.conf 2451
SNMPERR_BAD_ADDRESS 3074, 3088
SNMPERR_BAD_LOCPORT 3075
SNMPERR_BAD_SESSION 3070, 3088
SNMPERR_GENERIC 3075, 3081, 3088
snprintf() 3096
SO_BINDTODEVICE 969
SO_BROADCAST 969
SO_DEBUG 969
SO_DONTROUTE 969
SO_ERROR 970
SO_KEEPALIVE 970
SO_LINGER 966, 970, 2913
SO_OOBINLINE 971
SO_RCVBUF 971
SO_RCVLOWAT 971
SO_RCVTIMEO 966, 972, 2913
SO_REUSEADDR 972
SO_REUSEPORT 972
SO_SNDBUF 971
SO_SNDLOWAT 973
SO_SNDTIMEO 966, 973, 2913
SO_TIMESTAMP 973
SO_TYPE 974
SO_USELOOPBACK 974, 2622
SOCK_DGRAM 376, 973, 1114, 1400, 1425, 3101, 3102, 3587, 3616, 3811
SOCK_RAW 1054, 1056, 1400, 1401, 1406, 1423, 3101, 3102
SOCK_STREAM 123, 376, 1114, 1400, 1425, 1533, 2421, 2436, 3101, 3102, 3390, 3616
sockaddr_un 3616
sockatmark() 3099
sockcred 3617
SOCKCREDSIZE() 3618
socket() 1054, 1056, 1114, 1400, 1406, 1423, 2620, 2759, 3102, 3390, 3587, 3616
socketpair() 3105
sockets
  addresses
    errors 811
    freeing 756
    getting 816
    structure 133
connections
  accepting on 123, 2421
  initiating 376, 1910, 2438
  listening for 1532, 2615
  shutting down 2952
  status 1913
  creating 3102
  a pair of 3105
  connected 3105
datagrams 3101, 3102, 3587
debugging 969
file descriptors, testing for
  association 1445
managers, getting and setting
  information about 3338
messages
  peeking at 2483, 2486, 2490
  receiving from 2484, 2487, 2491
  sending to 2831, 2834, 2837
names
  binding to 241, 2433
  getting 964, 2608
options
  getting 967
  setting 2914
out-of-band (OOB) mark 3099
privileged IP port, binding
  to 244
privileged ports, getting 2629
raw 3101, 3102
stream 3101, 3102
types 3101
  determining 970, 974
SOCKS 3811
  classification 108
commands, executing
  remotely 2627
compiling for 3811
initializing 3108
library 3812
sockets
  connections 2421, 2438, 2615
  names 2433, 2608
socks3r.lib 3811
SOCKSinit() 3108, 3811
SOL_SOCKET 966, 2622
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sopen()</td>
<td>3111</td>
</tr>
<tr>
<td>sopenfd()</td>
<td>3116</td>
</tr>
<tr>
<td>sorting directory entries</td>
<td>155, 2674</td>
</tr>
<tr>
<td>quick sort</td>
<td>2417</td>
</tr>
<tr>
<td>space, amount free for a filesystem</td>
<td>778</td>
</tr>
<tr>
<td>space, filesystem</td>
<td>3200</td>
</tr>
<tr>
<td>space, testing a character for</td>
<td>1459, 1482</td>
</tr>
<tr>
<td>SPAWN_ALIGN_DEFAULT</td>
<td>3122</td>
</tr>
<tr>
<td>SPAWN_ALIGN_FAULT</td>
<td>3122</td>
</tr>
<tr>
<td>SPAWN_ALIGN_MASK</td>
<td>3124</td>
</tr>
<tr>
<td>SPAWN_ALIGN_NOFAULT</td>
<td>3122</td>
</tr>
<tr>
<td>SPAWN_CHECK_SCRIPT</td>
<td>3123</td>
</tr>
<tr>
<td>SPAWN_DEBUG</td>
<td>3123</td>
</tr>
<tr>
<td>SPAWN_EXEC</td>
<td>3123</td>
</tr>
<tr>
<td>SPAWN_EXPLICIT_SCHED</td>
<td>3123, 3125, 3126</td>
</tr>
<tr>
<td>SPAWN_FDCLOSED</td>
<td>3118, 3149</td>
</tr>
<tr>
<td>SPAWN_HOLD</td>
<td>3123</td>
</tr>
<tr>
<td>SPAWN_NEWPGROUP</td>
<td>3124</td>
</tr>
<tr>
<td>SPAWN_NOZOMBIE</td>
<td>3123</td>
</tr>
<tr>
<td>SPAWN_SEARCH_PATH</td>
<td>3123</td>
</tr>
<tr>
<td>SPAWN_SETGROUP</td>
<td>3120, 3123, 3124, 3151</td>
</tr>
<tr>
<td>SPAWN_SETND</td>
<td>3124</td>
</tr>
<tr>
<td>SPAWN_SETSID</td>
<td>3124</td>
</tr>
<tr>
<td>SPAWN_SETSIGDEF</td>
<td>3121, 3124, 3125, 3152</td>
</tr>
<tr>
<td>SPAWN_SETSIGIGN</td>
<td>3124, 3125</td>
</tr>
<tr>
<td>SPAWN_SETSIGMASK</td>
<td>3121, 3124, 3125, 3152</td>
</tr>
<tr>
<td>SPAWN_SETSTACKMAX</td>
<td>3124, 3125</td>
</tr>
<tr>
<td>SPAWN_TCSETPGROUP</td>
<td>3124</td>
</tr>
<tr>
<td>spawn()</td>
<td>3119</td>
</tr>
<tr>
<td>spawn* family of functions</td>
<td>50, 58, 1954</td>
</tr>
<tr>
<td>spawnl()</td>
<td>3130</td>
</tr>
<tr>
<td>spawnle()</td>
<td>3135</td>
</tr>
<tr>
<td>spawnlp()</td>
<td>3140</td>
</tr>
<tr>
<td>spawnlpe()</td>
<td>3145</td>
</tr>
<tr>
<td>spawnp()</td>
<td>3150</td>
</tr>
<tr>
<td>spawnp()</td>
<td>3150</td>
</tr>
<tr>
<td>spawnp()</td>
<td>3157</td>
</tr>
<tr>
<td>spawnve()</td>
<td>3162</td>
</tr>
<tr>
<td>spawnv()</td>
<td>3167</td>
</tr>
<tr>
<td>spawnvpe()</td>
<td>3171</td>
</tr>
<tr>
<td>SPD (Security Policy Database)</td>
<td>1410</td>
</tr>
<tr>
<td>special characters</td>
<td>1708</td>
</tr>
<tr>
<td>spinlocks</td>
<td></td>
</tr>
<tr>
<td>destroying</td>
<td>2363</td>
</tr>
<tr>
<td>initializing</td>
<td>2365</td>
</tr>
<tr>
<td>locking</td>
<td>2367, 2369</td>
</tr>
<tr>
<td>unlocking</td>
<td>2371</td>
</tr>
<tr>
<td>sporadic scheduling</td>
<td>2161</td>
</tr>
<tr>
<td>sprintf()</td>
<td>3175</td>
</tr>
<tr>
<td>spwd</td>
<td>2399</td>
</tr>
<tr>
<td>sqrt(), sqrtf()</td>
<td>3178</td>
</tr>
<tr>
<td>square roots</td>
<td>3178</td>
</tr>
<tr>
<td>srand()</td>
<td>3180</td>
</tr>
<tr>
<td>srand48()</td>
<td>3182</td>
</tr>
<tr>
<td>srandrandom()</td>
<td>3184</td>
</tr>
<tr>
<td>SS_REPL_MAX</td>
<td>2698</td>
</tr>
<tr>
<td>sscanf()</td>
<td>3189</td>
</tr>
<tr>
<td>st_atime</td>
<td>1654, 1657, 2067, 2448, 2476</td>
</tr>
<tr>
<td>st_atime</td>
<td>1654, 1657, 2067, 2448, 2476</td>
</tr>
<tr>
<td>st_ctime</td>
<td>317, 643, 646, 1509, 1523, 1654, 1657, 2067, 2617, 3791</td>
</tr>
<tr>
<td>st_fiame</td>
<td>1657, 2067</td>
</tr>
<tr>
<td>st_mode</td>
<td>3194</td>
</tr>
</tbody>
</table>
Index

stack  
  maximum size 2903, 3124
  memory, allocating from 152
  overflow, protecting against 2155
  threads 2142, 2144, 2148, 2165, 2167, 2171
stat  773, 3191
stat(), stat64() 1570, 3191
resource managers, implementing in 1358, 1360
STATE_CONDVAR 3306, 3506
STATE_INTR 3506
STATE_JOIN 3506
STATE_MUTEX 3305, 3306, 3506
STATE_RECEIVE 3506
STATE_REPLY 1817, 3506
STATE_SEM 3506
STATE_SEND 1817, 3506
STATE_SIGSUSPEND 3506
STATE_SIGWAITINFO 3506
static rules filter
  controlling 3540
statvfs, statvfs64 777, 3199
statvfs(), statvfs64() 3199
stderr 69, 650, 690, 2065, 3203
  buffering 2842, 2877
  command-line options, errors when parsing 907
daemons 2106
  formatted messages on 565, 3653, 3689, 3711
  host errors 1026
STDERR_FILENO 690, 3203
stdin 69, 650, 690, 3204
  characters, reading 675, 827, 829
daemons 2106
  input, formatted 2676, 3669, 3693, 3800
  strings, reading 954
  wide characters, reading 1005
STDIN_FILENO 690, 2067, 2077, 3204
stdout 69, 650, 690, 2397, 3205
  buffering 2842, 2877
  characters, writing 743, 2390, 2392
daemons 2106
  output, formatted 2096, 3667, 3691, 3787
  strings, writing 2397
  wide characters, writing 2409
STDOUT_FILENO 690, 2067, 2077, 3205
straddstr() 3206
stracecmp() 3208
strcat() 3211
strchr() 3213
strcspn() 3215
strcmpl() 3217
strcoll() 3219, 3288
strcpy() 3221
strcsnpl() 3223
strdup() 3225
Stream Control Transmission
  Protocol  See SCTP
stream I/O
  buffering
associating with 2840, 2926
block 2842
line 2877
characters
pushing back 3612, 3614
reading 673, 675, 679, 823,
825, 827, 829, 1003, 1005
writing 741, 743, 2386,
2388, 2390, 2392, 2407,
2409
closing 648, 650
end-of-file 666
clearing 328
errors 668
clearing 328
messages, printing 2065
file descriptors
associating streams with 663
getting 690
files
flushing 670, 705
locking 701, 792
opening 714
reading 1151
unlocking 797
input, formatted 766, 809,
2676, 3660, 3665, 3669,
3693, 3800
output, formatted 739, 804,
2096, 3657, 3663, 3667,
3691, 3787
pipes
closing 2063
creating 2067
opening 2076
position
getting 677, 783
setting 769, 771
reading 751
reopening 760
rewinding 2602
seeking 769
strings
reading 954
writing 745, 749, 2397
telling 783
temporary files 1664, 1666,
3522
wide characters
orientation 802
reading 685, 687
writing 747
words
getting next 1001
writing 2405
writing 806
stream sockets 3101, 3102
stream, returning to remote
command 2435, 2627
streams
flushing 3374
strerror() 3227
strftime() 3229
stricmp() 3235
strings See also characters; wide
characters
character, filling with 3253,
3263
comparing 237, 1627, 1633
case-insensitive 3208, 3217,
3235, 3241, 3251
case-sensitive 3215, 3246
locale’s collating sequence,
using 3219
concatenating 3206, 3211, 3244
configuration, getting and setting 371
copying 239, 1623, 3221, 3225, 3248, 3288
encrypting 406, 517, 2414, 2875
error messages 2065, 3227
formatted 3096, 3175, 3674, 3677
hexadecimal numbers,
   converting to/from 207
input, formatted 766, 2676, 3660, 3669
IP addresses, converting
to/from 1106, 1109
IPv4 addresses, converting
to/from 1085, 1087, 1102, 1104
length 3237
lowercase, converting to 3239
matching 711
network numbers, converting
to/from 1100
node descriptors, converting
to/from 1918, 1925
numbers, converting
to/from 205, 207, 209, 211, 1490, 1573, 3271, 3274, 3281, 3284, 3589, 3636
output, formatted 739, 2096, 3657, 3667
paths, resolving 2481
reversing 3259
scanning input from 3189, 3680, 3685
searching
   characters 1083, 2610, 3213, 3257
   sets of characters 3223, 3255, 3267
   sets of wide characters 3725, 3737, 3743
   slashes (/) 234, 438
   strings 3269
   wide characters 3717, 3739
signal descriptions 3265
splitting 3261, 3275, 3278
stdin, reading from 954
stdout, writing to 2397
streams
   reading from 679
   writing 745, 749
substrings
   comparing,
      case-insensitive 3241, 3251
   comparing,
      case-sensitive 3246
time_t, converting to/from 410
time, formatted 3229, 3727
tm, converting to/from 161
tokens, splitting into 3275, 3278
transforming 3288
uppercase, converting to 3286
zeroing 255
strlen() 3237
strlwr() 3239
strncasecmp() 3241
strncat() 3244
strncpy() 3248, 3288
strncmp() 3246, 3288
strnicmp() 3251
strnset() 3253
strrchr() 3257
strrev() 3259
strset() 3261
strset() 3263
strsignal() 3265
strspn() 3267
strstr() 3269
strtok() 3275
strtok(), strtok() 3281
strtoimax() 3284
strtokmax() 3274
strtok_r() 3278
sync() 3300
SYMLOOP_MAX 2472
resource managers,
implementing in 1365, 1367
SyncCondvarSignal(),
SyncCondvarSignal_r() 3301
SyncCondvarWait(),
SyncCondvarWait_r() 3304
SyncCtl(), SyncCtl_r() 3310
SyncDestroy(), SyncDestroy_r() 3312
synchronization objects See also
mutexes; semaphores;
threads
creating 3331
destroying 3312
mutexes
events 3310, 3314
locking 3317, 3322
priority 3310
reviving 3320
semaphores
incrementing 3325, 3327
threads
blocking 3304
waking up 3301
SyncMutexEvent(),
SyncMutexEvent_r() 3314
SyncMutexLock(),
SyncMutexLock_r() 3317
symbolic links
creating 3297
deleting 2509, 3619
information, getting 1570
ownership, changing 1508
reading 2471
resolving 2481
temporary 1987, 1989
symlink() 3297
SUN_LEN() 3616
swab() 3291
swprintf() 3293
swscanf() 3295
suboptions, parsing 984
SyncMutexRevive(),
  SyncMutexRevive_r() 3320
SyncMutexUnlock(),
  SyncMutexUnlock_r() 3322
SyncSemPost(), SyncSemPost_r() 3325
SyncSemWait(), SyncSemWait_r() 3327
SyncTypeCreate(),
  SyncTypeCreate_r() 3331
sys_errlist 569
sys_err 569
sys_nsig 2988
sys_siglist 2988
sysconf() 3334
sysctl() 3338
syslog() 3345
sysmgr_reboot() 3348
SYSPAGE_CPU_ENTRY() 3350
SYSPAGE_ENTRY() 3352
qtime 3352
  boot_time 3352
  cycles_per_sec 3352
system
clock
  getting 358
  period, getting and setting 355
  setting 358
  ticks per second 3352
commands, executing 3356
daemons 413, 2106
termination, notification of 2108
notification of 2109
triggering 2113
hardware information 1042, 1044, 1045, 1047
instruction set architecture 371
limits, getting 3334
rebooting 3348
rebooting on bankruptcy 2719
resources
  creating 2640
  destroying 2644
  limits, getting 836, 946
  limits, setting 2902
  querying 2655
  reserving 2633
  returning 2646
  usage, getting 949
time since booting 3352
time, adjusting 347
System Analysis Toolkit (SAT) 1180
system databases
groups
  closing 520
  ID, getting information about 849, 851
  membership 1146
  name, getting information about 854, 856
  next entry, getting 846
 rewinding 2859
passwords
  closing 548
  encrypting 406, 2414
  entry, getting for a user 935, 938, 940, 943
  entry, getting next 932
rewinding 2895
shadow passwords
   closing 551
   entry, reading 682, 977, 981
   entry, structure 2399
   entry, writing 2399
   rewinding 2916
system message log
   closing 366
   log priority mask 2882
   opening 1962
   writing to 3345, 3687
   blocks 3060
   formatted output 3063, 3673
   integers 3066
system packet forwarding database
   See ROUTE
system page 3355
   CPU-specific entry, getting a
      pointer to 3350
   entry, getting a pointer to 3352
System partition 2721
system() 50, 3356

T
T_PTR 2518, 2521, 2524, 2527
tan(), tanf() 3362
tangents 3362
   hyperbolic 3364
   inverse hyperbolic 200
tanh(), tanhf() 3364
target operating system 113
tcdrain() 3366
tcdropline() 3368
tcflow() 3371
tcflush() 3374
tcgetattr() 3377
tcgetpgrp() 3379
tcgetsid() 3381
tcgetsiz() 3383
TCI_FLUSH 3374
tcinject() 3385
TCIOFF 3372
TCIOFFHW 3372
TCIO.Flush 3374
TCION 3372
TCIONHW 3372
tciscchars() 3388
TCOFF 3374
TCOFF HW 3374
TCP (Transmission Control
   Protocol) 3390
   connection, closing 521
   SOCKS 3812
TCP_KEEPALIVE 3391
TCP_MAXSEG 3391
TCP_NODELAY 974, 3391
TCP/IP
   address information
      addrinfo 133
   addresses
      manipulating 1091
      network numbers,
         extracting 1096
      strings, converting
to/from 1085, 1087, 1102,
      1104, 1106, 1109
   errors 1020, 1026, 1035
host entries
  getting  870, 874, 876, 879
hosts database
  opening  2863
Internet domain names
  compressing  484
  expanding  486
messages
  receiving  2484, 2487, 2491
  sending  2831, 2834, 2837
network database
  closing  546
  opening  2884
protocols database
  closing  547
  opening  2893
services database
  closing  550
  entry structure  2839
  opening  2909
sockets
  ports, binding to  244
TCSADRAIN  3396
TCSAFLUSH  3396
TCSANOW  3396
tcsendbreak()  3393
tcsetattr()  3396
tcsetpgrp()  3399
tcsetsid()  3402
tcsetsize()  3404
tell()  1568
tell(), tell64()  3406
telldir()  3409
tempnam()  3411
temporary files
  creating  3522
  creating and opening  1664
name, generating  1666, 3411, 3525
TERM  325
terminal control
  characters, injecting  3385
  communications line
    break condition, asserting  3393
    disconnecting  3368
    draining  3366
    flow control  3371
    flushing  3374
    process group ID  3381
    getting  3379, 3399
    size  3383, 3404
terminal group, starting new  3124
terminals
  attributes, setting  290
  canonical input buffer  736, 1980
control
  attributes  3377, 3396
  structure  3413
controlling
  making  3402
  path name  408
file descriptor, testing for
  association with  1439
input speed  279, 294
operating attributes  1210
output speed  282, 297
raw input buffer  736, 1980
reading  2458
TERMINFO  325
termios  290, 2458, 3396, 3413
text segment
  beginning of  252
end of 576
thread_pool_attr_t 3421
THREAD_POOL_CONTROL_HIWATER 3417
THREAD_POOL_CONTROL_INCREMENT 3417
THREAD_POOL_CONTROL_LOWATER 3417
THREAD_POOL_CONTROL_MAXIMUM 3418
THREAD_POOL_CONTROL_NONBLOCK 3418
thread_pool_control() 3418
thread_pool_create() 3420
thread_pool_destroy() 3427
thread_pool_limits() 3431
thread_pool_start() 3433
ThreadCancel(), ThreadCancel_r() 3436
ThreadCreate(), ThreadCreate_r() 3441
ThreadCtl(), ThreadCtl_r() 1155, 1164, 1169, 1172, 1174, 1182, 1184, 1187, 1189, 3447
ThreadDestroy(), ThreadDestroy_r() 3455
ThreadDetach(), ThreadDetach_r() 3458
ThreadJoin(), ThreadJoin_r() 3460
threads
aborting 2124
attributes 3441
contention scope 2140, 2163
destroying 2128
detach state 2130, 2153
guard area, size of 2132, 2155
initializing 2150
scheduling parameters 2136, 2159
scheduling policy 2138, 2161
stack address 2142, 2165
stack size 2148, 2171
stack, lazy 2144, 2167
bankruptcy, notifying of 2737
barriers
attributes 2175, 2179, 2183
attributes, process-shared 2181, 2185
destroying 2173
initializing 2175
waiting at 2177
blocking 3304
busy-waiting 1896, 1901, 1903, 1905
calibrating 1898
canceling 2187, 3436
cancellation
cleanup handlers 2189, 2191
points 3444
points, creating 2373
state 2333
type 2335
clock ID 2235
concurrency 2233, 2337
condition variables
attributes 2210, 2216
attributes, clock 2212, 2218
attributes,
  process-shared 2214, 2220
blocking on 2202, 2206
destroying 2196, 3312
initializing 2198, 3331
unblocking 2194, 2200
creating 2223, 3441
critical
  determining if 2736
  marking as 2734, 2973
data 2241, 2245, 2249, 2343, 3444
destroying 3455
detached 2153, 3443
detaching from a process 2227, 3458
errno 568, 3444
events
  attaching 2737
files, locking 701
fork handlers, registering 2126
freezing 3451
I/O privileges, requesting 3449
IDs
  calling thread 2332
  comparing 2229
initializing 2300
joinable 2153, 3443
joining 2243, 3460
  with a time limit 2374
keys 2245, 2249
local storage 2241, 2245, 2249, 2343, 3444
maximum number of 2903
misaligned access
  response 3447
mutexes
  attributes, destroying 2272
  attributes, initializing 2286
  attributes, priority ceiling 2274, 2289
  attributes, process-shared 2279, 2293
  attributes, recursive 2281, 2295
  attributes, scheduling protocol 2276, 2292
  attributes, type 2283, 2298
destroying 2253, 3312
initializing 2257, 3331
locking 2259, 2265, 2268
priority ceiling 2255, 2263
unlocking 2270
names 2237, 2339
noncritical, marking as 2735, 2973
once-initialization 2300
partitions
determining 2736
joining 2726, 2743
pool See also resource managers
  attributes, changing 3418, 3431
  creating 3420
destroying 3427
starting 3433
priority on receiving a pulse 2969
private data 2241, 2245, 2249, 2343, 3444
processor affinity 3450
### read-write locks
- **attributes, creating** 2328
- **attributes, destroying** 2324
- **attributes, process-shared** 2326, 2330
- **destroying** 2303, 3312
- **initializing** 2305, 3331
- **locking for reading** 2308, 2310, 2316
- **locking for writing** 2313, 2318, 2322
- **unlocking** 2320
- **return status** 3442
- **scheduling parameters** 2239, 2342, 2747, 2754
- **scheduling policy** 2239, 2342, 2747, 2754, 3442
- **inherit** 2134, 2157, 3443
- **scope** 3443
### signal mask
- **getting** 3019
- **restoring** 2979
- **saving** 3025
- **setting** 3019, 3029
- **signal-blocked** 3002
### signals
- **initial state** 3444
- **mask** 2345
- **sending** 2251
- **targeting** 2957, 2993, 2996
- **terminating on** 3443
- **waiting for** 1991
### sleepon locks
- **destroying** 3048
- **initializing** 3050
- **locking** 2349, 3052
- **unlocking** 2347, 2351, 3046, 3054
- **unlocking** 2357, 3056
- **waiting** 2353, 2359, 3058
### spinlocks
- **destroying** 2363
- **initializing** 2365
- **locking** 2367, 2369
- **unlocking** 2371
- **stack** 3442
- **suspending** 340, 1894, 3029, 3044
- **synchronizing** 2177
- **system overload, notifying of** 2737
- **terminating** 2231
- **terminating** unconditionally 2124
- **unfreezing** 3451
- **waking up** 2347, 2351, 3046, 3054, 3301
- **yielding** 2710, 2757
- **zombies** 3443
- **tick size, changing** 355, 2715
- **ticksize, getting and setting** 355

### time_t 3463
- **tm, converting to/from** 1016, 1018
### time members
- **in attribute structure of resource managers** 1222
- **Time Stamp Counter (TSC)** 349
- **time to live (TTL)** 1400
### time zone
- **abbreviations** 3580
- **default** 372
- **offset from UTC** 3519
setting 3581
time() 3463
timeb 786
TIMED_OUT 3092, 3094
timeout, setting on a blocking state 3481
timeouts, SNMP 3094
TIMER_ABSTIME 339, 3477
timer_create() 3465
timer_delete() 3469
timer_getexpstatus() 3471
timer_getoverrun() 3473
timer_gettime() 3475
timer_settime() 3478
timer_timeout(), timer_timeout_r() 3481
TimerAlarm(), TimerAlarm_r() 3489
TimerCreate(), TimerCreate_r() 3492
TimerDestroy(), TimerDestroy_r() 3495
TimerInfo(), TimerInfo_r() 3498
timers
  alarm, scheduling 3489
  creating 3492
  destroying 3495
  information about, getting 3498
  interval
    value, getting 889
    value, setting 2869
realtime
  creating 3465
  destroying 3469
  expiry status 3471
  overruns 3473
time until expiry 3475, 3478, 3502
  threads 340
  timeout, setting on kernel blocking state 3506
TimerSettime(), TimerSettime_r() 3502
TimerTimeout(), TimerTimeout_r() 3506
times
  booting, since 3352
  calendar
    current 3463
    local, converting to/from 1539, 1541, 1668
    structure 3520
  clock
    adjusting 347
    cycles 349
    getting 336
    getting and setting 358
    ID, getting 332, 351
    period, getting and setting 355
    resolution, getting 334
    setting 343
  current calendar 3463
    getting 786, 989
    setting 2919
daylight saving time 415, 3581
difference, calculating 433
files
  access 1370, 1382, 1385, 3629
  modification 799, 1370, 1382, 1385, 3626, 3629
Index

status-change 1370, 1382, 1385
formatting 2880, 3229, 3727
local calendar, converting to/from 1539, 1541, 1668
nanoseconds
timespec, converting to/from 1938, 3517
processes execution time limit, getting 2702
execution time, in clock ticks 330
specification structure 3516
time_t strings, converting to/from 1016, 1018
timespec nanoseconds, converting to/from 1938, 3517
tm strings, converting to/from 161
times() 3513
timespec 334, 3516
nanoseconds, converting to/from 1938, 3517
timespec2nsec() 3517
timezone 3519, 3581
tm 1016, 1539, 1668, 3520, 3727
strings, converting to/from 161, 410
time_t, converting to/from 1016, 1018
TMP_MAX 3411
tmpfile() 3522
tmpnam() 3525
tms 3513
tokens, breaking a string into 3275, 3278, 3752
tolower() 3528
TOS (type of service) 1400
TOSTOP 3415
toupper() 3530
towctrans() 3532
towlower() 3534
towupper() 3536
TraceEvent() 3540, 3547
traceparser_cs() 3560
TRACEPARSER_INFO_* 3567
traceparser() 3558
Transmission Control Protocol See TCP
trigonometry See also hyperbolic functions
arccosine 129
arcsine 164
arctangent 196, 198
cosine 398
sine 3040
tangent 3362
TRP_REQ_MSG 3080
TRP2_REQ_MSG 3080
truncate() 3573
TRYAGAIN 1021, 1027
TSC (Time Stamp Counter) 349
TTL (time to live) 1400
ttyname_r() 3578
ttyname() 3576
type of service (TOS) 1400
typographical conventions liii
Index

TZ 325, 3581
tzname 3580, 3581
tzset() 1668, 3232, 3581

U
ualarm() 3584
UDP (User Datagram Protocol) 3587
    not supported by SOCKS 3811
UIO_MAXIOV 2474, 2477, 3797
ulltoa() 3589
utoa() 3589
umask() 3592
umount() 3595
UNALIGNED_PUT16() 3597
UNALIGNED_PUT32() 3599
UNALIGNED_PUT64() 3601
UNALIGNED_RETI16() 3603
UNALIGNED_RETI32() 3605
UNALIGNED_RETI64() 3607
uname() 3609
ungetc() 3612
ungetwc() 3614
Unicode 99
union, offset of members within 1944
Unix classification 108
UNIX-domain protocol 3616
unlink() 3619
unsetenv() 3622
uppercase
    characters, converting to 3530
    strings, converting to 3286
testing a character for 1462, 1484
wide characters, converting to 3532, 3536, 3770
usage of system resources 949
USER_PROCESS 995, 3633
User Datagram Protocol (UDP) 3587
    not supported by SOCKS 3811
user information file
    closing 553
    entry 3632
    reading 993, 998
    renaming 3634
    returning to beginning of 2924
    searching 995
    writing 2402
users
    IDs
        effective 842, 2853, 2899, 2921
        real 991, 2899, 2921
        saved 2921
        set-user 3335
        names 891, 893
        password database, getting entry for 935, 938, 940, 943
    processes, maximum per real user ID 2903, 3334
usleep() 3624
utilities, locating 372
utimbuf 799, 3626
utime() 3626
    resource managers, implementing in 1382, 1385
utimes() 3629
Index

utmp 3632
utmpname() 3634
utoa() 3636
utsname 3609

V

va_arg() 3639
va_copy() 3645
va_end() 3647
va_start() 3649
valloc() 3651
variable_list 3078
variable-length argument lists
   ("varargs") 3639, 3645, 3647, 3649
   coercion 3640
variables, global
   __progname 2122
   __ambksz 157
   __argc 159
   __argv 160
   __auxv 233
   __btext 252
   __edata 516
   __end 519
   __etext 576
   __syspage_ptr 3355
daylight 415
erro 568
optarg 906
opterr 907
optind 906
optopt 907
stderr 3203
stdin 3204
stdout 3205
sys_errlist 569
sys_errno 569
sys_nsig 2988
sys_siglist 2988
timezone 3519
tzname 3580
verr(), verrx() 3653
vfork() 3655
vfprintf() 3657
vfscanf() 3660
vfsspfmtf() 3663
vfsspfmtf() 3665
video memory, sharing 1682
virtual 8086 mode 1194
void pointers, size of 113
vprintf() 3667
vsscanf() 3669
vsscanf() 3673
vsscanf() 3674
vsscanf() 3677
vsscanf() 3680
vsscanf() 3683
vsscanf() 3685
vsscanf() 3687
vsscanf() 3689
vsscanf() 3691
vsscanf() 3693

W

W_OK 126, 513
wait() 3514, 3695, 3702, 3709
wait3() 3699
wait4()  3702
waitid()  3706
waitpid()  3514, 3709
warn(), warnx()  3711
warnings, formatted on stderr  3689, 3711
WCONTINUED  3698, 3701, 3705, 3708
WCOREDUMP()  3696
wctomb()  3767
wcstrans()  3770
wctypename()  3772
WEXITED  3698, 3701, 3705, 3708
WEXITSTATUS()  3357, 3696
whitespace, testing a character for  1459, 1482
wide characters  See also
characters; strings
classes  3772
copying  3532, 3770
overlapping objects  3780
lowercase, converting to  3532, 3534, 3770
multibyte characters, converting to/from  1602, 1612, 3713, 3767
conversion object, status of  1605
searching
in a string  3717, 3739
in memory  3774
sets of, searching for  3725, 3737, 3743
setting  3782
single-byte characters, converting to/from  253, 3765
stdin, reading from  1005
stdout, writing to  2409
streams
orientation  802
pushing back  3614
reading  685, 687, 1003
writing to  747, 2407
strings
Index

© 2007, QNX Software Systems GmbH & Co. KG.

comparing 3719, 3721, 3733, 3776
concatenating 3715, 3731
copying 3723, 3735, 3763
formatted 3293, 3683
input, formatted 809, 3295, 3665, 3693, 3800
length 3729
multibyte characters,
  converting to/from 1607, 1609, 3741, 3757
numbers, converting
to/from 3747, 3751, 3755, 3761
output, formatted 804, 3663, 3691, 3787
searching for a set of wide characters 3725, 3737, 3743
searching for a string 3745
searching for a wide character 3717, 3739
splitting into tokens 3752
streams, writing to 749
transforming 3763
testing for
  alphabetic 1466
  alphanumeric 1464
  character class 1470
  control character 1468
decimal digit 1472
hexadecimal digit 1486
lowercase 1476
printable 1474, 1478
punctuation 1480
uppercase 1484
whitespace 1482

uppercase, converting to 3532, 3536, 3770
wide mode
  controlling 3540
WIFCONTINUED() 3696
WIFEXITED() 3696
WIFSIGNALED() 3696
WIFSTOPPED() 3696
wmemchr() 3774
wmemcmp() 3776
wmemcpy() 3778
wmemmove() 3780
wmemset() 3782
WNOHANG 3698, 3701, 3705, 3708
WNOWAIT 3698, 3701, 3705, 3708
word expansions 3784, 3786
wordexp() 3784
wordfree() 3786
working directory 309, 831, 1007
wprintf() 3787
WRDE_NOSYS 3784
write() 1222, 3789
  resource managers,
    implementing in 1388
writeblock() 3794
writev() 3797
wscanf() 3800
WSTOPPED 3698, 3702, 3705, 3709
WSTOPSIG() 3696
WTERMSIG() 3696
WUNTRACED 3698, 3702, 3705, 3709
X

X_OK 126, 513

Y

y0(), y0f() 3802
y1(), y1f() 3804
yn(), ynf() 3806

Z

zombies
  preventing children from
    becoming 2956, 2991, 3123
  threads 3443