QNX® Neutrino® Realtime Operating System

Library Reference — A to E

For QNX® Neutrino® 6.3

© 2006, QNX Software Systems GmbH & Co. KG.
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
</table>

**About This Reference**  lii

- Typographical conventions  liii
- Note to Windows users  liv

**What you’ll find in this guide**  lv

**What’s new in QNX Neutrino 6.3.2**  lv

- New Content  lv
- Changed Content  lvi

**What’s new in QNX Neutrino 6.3.0 Service Pack 2**  lviii

- New Content  lviii
- Changed Content  lviii
- Errata  lix

**What’s new in QNX Neutrino 6.3.0 Service Pack 1**  lix

- New content  lix
- Changed content  lx
- Errata  lxii

**What’s new in QNX Neutrino 6.3.0**  lxiii

- New content  lxiii

**What’s new in QNX Neutrino 6.2.1**  lxiv

- New content  lxiv
- Changed content  lxv
- Errata  lxv

**What’s new in QNX Neutrino 6.2**  lxvii

- New Content  lxvii
- Deprecated Content  lxx
- Errata  lxx

November 2, 2006
What’s new in the QNX Neutrino 6.1.0 docs lxx
   New content lxxi
   Deprecated content lxxii

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 37
  Operating system I/O functions 40
  PC Card functions 44
  Platform-specific functions 44
  Process environment functions 46
  Process manipulation functions 49
  Realtime timer functions 58
  Resource manager functions 60
  Searching and sorting functions 65
  Shared memory functions 67
  Signal functions 67
  Stream I/O functions 69
  String manipulation functions 72
System database functions 75
System message log functions 76
TCP/IP functions 77
Terminal control functions 85
Thread functions 86
Time functions 95
Variable-length argument list functions 98
Wide-character functions 98

What’s in a function description? 102

Synopsis: 102
Arguments: 103
Library: 103
Description: 103
Returns: 103
Errors: 103
See also: 104
Examples: 104
Classification: 104
Function safety: 108

Manifests 111

QNX Neutrino Functions and Macros 115

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort()</td>
<td>119</td>
</tr>
<tr>
<td>abs()</td>
<td>121</td>
</tr>
<tr>
<td>accept()</td>
<td>123</td>
</tr>
<tr>
<td>access()</td>
<td>126</td>
</tr>
<tr>
<td>acos(), acosf()</td>
<td>129</td>
</tr>
<tr>
<td>acosh(), acoshf()</td>
<td>131</td>
</tr>
<tr>
<td>addrinfo</td>
<td>133</td>
</tr>
<tr>
<td>aio_cancel()</td>
<td>135</td>
</tr>
<tr>
<td>aio_error()</td>
<td>137</td>
</tr>
<tr>
<td>aio_fsync()</td>
<td>139</td>
</tr>
<tr>
<td>aio_read()</td>
<td>141</td>
</tr>
</tbody>
</table>
aio_return()  143
aio_suspend()  145
aio_write()  147
alarm()  149
alloca()  152
alphasort()  155
_amblksiz  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
atoh()  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 234
basename() 235
bcmp() 238
bcopy() 240
bind() 242
bindresvport() 245
brk() 247
bsearch() 250
_btext 253
btowc() 254
bzero() 256
cabs(), cabsf() 258
cache_fini() 260
CACHE_FLUSH() 262
cache_init() 265
CACHE_INVALID() 271
calloc() 274
cbrt(), cbrtf() 276
ceil(), ceilf() 278
cfgetispeed() 280
cfgetospeed() 283
cfgopen() 286
cfmakera()  290
 cfree()  292
 cfsetispeed()  294
 cfsetospeed()  297
 ChannelCreate(), ChannelCreate_r()  300
 ChannelDestroy(), ChannelDestroy_r()  307
 chdir()  310
 chmod()  313
 chown()  317
 chroot()  320
 chsize()  323
 clearenv()  326
 clearer()  329
 clock()  331
 clock_getcpu_clockid()  333
 clock_getres()  335
 clock_gettime()  337
 clock_nanosleep()  340
 clock_settime()  344
 ClockAdjust(), ClockAdjust_r()  347
 ClockCycles()  350
 ClockId(), ClockId_r()  353
 ClockPeriod(), ClockPeriod_r()  356
 ClockTime(), ClockTime_r()  360
 close()  363
 closedir()  365
 closelog()  368
 _cmdfd()  369
 _cmdname()  370
 confstr()  373
 connect()  378
 ConnectAttach(), ConnectAttach_r()  381
ConnectClientInfo(), ConnectClientInfo_r()  386
ConnectDetach(), ConnectDetach_r()  390
ConnectFlags(), ConnectFlags_r()  392
ConnectServerInfo(), ConnectServerInfo_r()  395
copysign(), copysignf()  398
cos(), cosf()  400
cosh(), coshf()  402
creat(), creat64()  404
crypt()  408
ctermid()  410
ctime(), ctime_r()  412
daemon()  415
daylight  417
DebugBreak()  418
DebugKDBreak()  420
DebugKDOutput()  421
delay()  423
devctl()  425
difftime()  435
dircntl()  437
dirname()  440
dispatch_block()  443
dispatch_context_alloc()  447
dispatch_context_free()  450
dispatch_create()  452
dispatch_destroy()  456
dispatch_handler()  459
dispatch_timeout()  463
dispatch_unblock()  466
div()  468
dladdr()  470
dlclose()  473
dlerror() 475
dlopen() 477
dlsym() 484
dn_comp() 487
dn_expand() 489
drand48() 491
drem(), dremf() 493
ds_clear() 495
ds_create() 497
ds_deregister() 500
ds_flagst() 502
ds_get() 504
ds_register() 506
ds_set() 508
dup() 510
dup2() 513
eaccess() 516
_edata 519
encrypt() 520
_end 522
endgrent() 523
endhostent() 525
ENDIAN_BE16() 527
ENDIAN_BE32() 529
ENDIAN_BE64() 531
ENDIAN_LE16() 533
ENDIAN_LE32() 535
ENDIAN_LE64() 537
ENDIAN_RET16() 539
ENDIAN_RET32() 541
ENDIAN_RET64() 543
ENDIAN_SWAP16() 545
ENDIAN_SWAP32() 547
ENDIAN_SWAP64() 549
endnetent() 551
endprotoent() 552
endpwent() 554
endservent() 556
endspent() 557
endutent() 559
environ 561
eof() 562
erand48() 565
erf(), erff() 567
erfc(), erfcf() 569
erf(), erfx() 571
errno 574
__text 583
execl() 584
execle() 591
execlp() 598
execle() 604
execv() 609
execve() 615
execvp() 621
execvpe() 627
_exit() 632
exit() 635
exp(), expf() 638
expm1(), expm1f() 640
C Library — F to H 643
fabs(), fabsf() 646
fcfgopen() 648
fchmod() 650
fchown() 653
fclose() 656
fcloseall() 658
fcntl() 660
fdatasync() 669
fdopen() 671
feof() 674
ferror() 676
fflush() 678
ffs() 680
fgetc() 681
fgetchar() 683
fgetpos() 685
fgets() 687
fgetspent() 690
fgetwc() 693
fgetws() 695
fileno() 698
finite(), finitef() 701
flink() 703
flock() 706
flockfile() 709
floor(), floorf() 711
flushall() 713
fmod(), fmodf() 715
fnmatch() 718
fopen(), fopen64() 722
fork() 727
forkpty() 731
fp_exception_mask() 733
fp_exception_value() 736
fp_precision() 739
fp_rounding()  742
fpathconf()  745
fprintf()  748
fputc()  750
fputchar()  752
fputs()  754
fputwc()  756
fputws()  758
fread()  760
free()  763
freeaddrinfo()  765
freefaddr()  767
freopen(),freopen64()  769
frexp(),frexpf()  773
fscanf()  775
fseek(),fseeko()  777
fsetpos()  780
fstat(),fstat64()  782
fstatvfs(),fstatvfs64()  786
fsync()  790
ftell(),ftello()  792
ftime()  795
ftruncate(),ftruncate64()  798
ftrylockfile()  801
ftw(),ftw64()  803
funlockfile()  806
futime()  808
fwide()  811
fwprintf()  813
fwrite()  815
fscanf()  818
gai_strerror()  820
gamma(), gamma_r(), gammaf(), gammaf_r() 822
getaddrinfo() 825
getc() 832
getc_unlocked() 834
getchar() 836
getchar_unlocked() 838
getcwd() 840
getdomainname() 843
getdtablesize() 845
getegid() 847
getenv() 849
geteuid() 851
getgid() 853
getgrent() 855
getgrgid() 858
getgrgid_r() 860
getgrnam() 863
getgrnam_r() 865
getgrouplist() 868
getgroups() 871
gethostbyaddr() 873
gethostbyaddr_r() 876
gethostbyname(), gethostbyname2() 879
gethostbyname_r() 882
gethostent() 885
gethostent_r() 887
gethostname() 890
getifaddrs() 892
GETIOVBASE() 894
GETIOVLEN() 896
getitimer() 898
getlogin() 901
getlogin_r()  903
getnameinfo()  905
getnetbyaddr()  910
getnetbyname()  912
getnetent()  914
getopt()  916
getpass()  922
getpeernam()  924
getpgid()  926
getpgpr()  928
getpid()  930
getppid()  932
getprio()  934
getprotobyname()  936
getprotobynumber()  938
getprotoent()  940
getpwent()  942
getpwnam()  945
getpwnam_r()  947
getpwuid()  950
getpwuid_r()  952
getrlimit(), getrlimit64()  955
getusage()  959
gets()  964
getservbyname()  966
getservbyport()  968
getservent()  970
getsid()  972
getsockname()  974
getsockopt()  976
getspent(), getspent_r()  987
getspnam(), getspnam_r()  991
getsubopt()  994
gettimeofday()  999
getuid()  1001
getutent()  1003
getutid()  1005
getutline()  1008
getw()  1011
getwc()  1013
getwchar()  1015
getwd()  1017
glob()  1019
globfree()  1024
gmtime()  1026
gmtime_r()  1028
h_errno  1030
hcreate()  1032
hdestroy()  1034
herror()  1036
hostent  1039
hsearch()  1041
hstrerror()  1045
htonl()  1047
htons()  1049
hwi_find_item()  1051
hwi_find_tag()  1054
hwi_off2tag()  1056
hwi_tag2off()  1058
hypot(), hypotf()  1060

C Library — I to L  1063
ICMP  1066
ICMP6  1068
if_freenamindex()  1071
if_indextoname() 1073
if_nameindex() 1075
if_nametoindex() 1077
ifaddrs 1079
ilogb(), ilogbf() 1081
in8() 1083
in8s() 1085
in16(), inbe16(), inle16() 1087
in16s() 1089
in32(), inbe32(), inle32() 1091
in32s() 1093
index() 1095
inet_addr() 1097
inet_aton() 1099
inet_ipn事故发生() 1101
inet_makeaddr() 1103
inet_net_ntop() 1105
inet_netof() 1108
inet_net_phton() 1110
inet_network() 1112
inet_ntoa() 1114
inet_ntoa_r() 1116
inet_ntop() 1118
inet_phton() 1122
INET6 1127
inet6_option_alloc() 1131
inet6_option_append() 1133
inet6_option_find() 1135
inet6_option_init() 1137
inet6_option_next() 1139
inet6_option_space() 1141
inet6_rthdr_add() 1143
inet6_rthdr_getaddr() 1145
inet6_rthdr_getflags() 1147
inet6_rthdr_init() 1149
inet6_rthdr_lasthop() 1151
inet6_rthdr_reverse() 1153
inet6_rthdr_segments() 1155
inet6_rthdr_space() 1157
initgroups() 1159
initstate() 1161
input_line() 1164
InterruptAttach(), InterruptAttach_r() 1167
InterruptAttachEvent(), InterruptAttachEvent_r() 1176
InterruptDetach(), InterruptDetach_r() 1182
InterruptDisable() 1185
InterruptEnable() 1187
InterruptHookIdle() 1189
InterruptHookTrace() 1193
InterruptLock() 1195
InterruptMask() 1198
InterruptUnlock() 1201
InterruptUnmask() 1203
InterruptWait(), InterruptWait_r() 1205
_intr_v86() 1208
_io_connect 1212
_io_connect_ftype_reply 1219
_io_connect_link_reply 1221
ioctl() 1224
iofdinfo() 1226
iofunc_attr_init() 1228
iofunc_attr_lock() 1230
iofunc_attr_t 1232
iofunc_attr_trylock() 1238
iofunc_attr_unlock() 1240
iofunc_check_access() 1242
iofunc_chmod() 1246
iofunc_chmod_default() 1249
iofunc_chown() 1251
iofunc_chown_default() 1254
iofunc_client_info() 1256
iofunc_close_dup() 1258
iofunc_close_dup_default() 1261
iofunc_close_ocb() 1263
iofunc_close_ocb_default() 1265
iofunc_devctl() 1267
iofunc_devctl_default() 1271
iofunc_fdinfo() 1273
iofunc_fdinfo_default() 1276
iofunc_func_init() 1280
iofunc_link() 1283
iofunc_lock() 1287
iofunc_lockcalloc() 1289
iofunc_lock_default() 1291
iofunc_lock_free() 1294
iofunc_lock_ocb_default() 1296
iofunc_lseek() 1298
iofunc_lseek_default() 1301
iofunc_mknod() 1303
iofunc_mmap() 1306
iofunc_mmap_default() 1310
iofunc_notify() 1312
iofunc_notify_remove() 1319
iofunc_notify_trigger() 1321
iofunc_ocb_attach() 1324
iofunc_ocbcalloc() 1326
iofunc_ocb_detach() 1329
iofunc_ocb_free() 1332
  iofunc_ocb_t 1334
iofunc_open() 1337
iofunc_open_default() 1343
iofunc_openfd() 1346
iofunc_openfd_default() 1350
iofunc_pathconf() 1352
iofunc_pathconf_default() 1355
iofunc_read_default() 1357
iofunc_read_verify() 1359
iofunc_readlink() 1363
iofunc_rename() 1366
iofunc_space_verify() 1370
iofunc_stat() 1374
iofunc_stat_default() 1376
iofunc_sync() 1379
iofunc_sync_default() 1381
iofunc_sync_verify() 1383
iofunc_time_update() 1386
iofunc_unblock() 1388
iofunc_unblock_default() 1390
iofunc_unlink() 1393
iofunc_unlock_ocb_default() 1396
iofunc_utime() 1398
iofunc_utime_default() 1401
iofunc_write_default() 1404
iofunc_write_verify() 1406
ionotify() 1410
IP 1416
IPsec 1423
ipsec_dump_policy() 1431
ipsec_get_policylen() 1433
ipsec_set_policy() 1435
ipsec_strerror() 1440
IP6 1442
isalnum() 1452
isalpha() 1454
isascii() 1456
isatty() 1458
iscntrl() 1460
isdigit() 1462
isfdtype() 1464
isgraph() 1466
isinf(), isnanf() 1468
islower() 1470
isnan(), isnanf() 1472
isprint() 1474
ispunct() 1476
isspace() 1478
isupper() 1481
iswalnum() 1483
iswalpha() 1485
iscntrl() 1487
iswctype() 1489
ishdigit() 1491
iswgraph() 1493
iswlower() 1495
iswprint() 1497
iswpunct() 1499
iswspace() 1501
iswupper() 1503
iswxdigit() 1505
isxdigit() 1507
itoa() 1509
j0(), j0f() 1512
j1(), j1f() 1514
jn(), jnf() 1516
jrand48() 1518
kill() 1520
killpg() 1523
labs() 1525
lchown() 1527
lcong48() 1530
ldexp(), ldexpf() 1532
ldiv() 1534
lfind() 1536
lgamma(), lgamma_r(), lgammaf(), lgammaf_r() 1539
link() 1543
lio_listio() 1547
listen() 1552
localeconv() 1554
localtime() 1559
localtime_r() 1561
lockf() 1563
log(), logf() 1567
log1p(), log1pf() 1569
log10(), log10f() 1571
logb(), logbf() 1573
login_tty() 1576
longjmp() 1578
lrand48() 1581
lsearch() 1583
lseek(), lseek64() 1586
lstat(), lstat64() 1590
ltoa(), lltoa() 1593
ltrunc() 1596

C Library — M to O 1601

main() 1604
mallinfo() 1607
malloc() 1609
mallopt() 1612
max() 1616
mblen() 1618
mbrlen() 1621
mbrtowc() 1623
mbsinit() 1626
mbsrtowcs() 1628
mbstowcs() 1630
mbtowc() 1633
mcheck() 1636
mem_offset(), mem_offset64() 1639
memalign() 1643
memccpy() 1645
memchr() 1647
memcpy() 1650
memcpyv() 1653
memcpys() 1655
memcmp() 1657
memmove() 1660
memset() 1662
message_attach() 1664
message_connect() 1671
message_detach() 1674
min() 1677
mkdir() 1679
mkfifo() 1682
mknod() 1685
mkstemp() 1689
mktemp() 1691
mktime() 1693
mlock() 1696
mlockall() 1699
mmap(), mmap64() 1703
mmap_device_io() 1710
mmap_device_memory() 1712
modem_open() 1716
modem_read() 1721
modem_script() 1725
modem_write() 1733
modff(), modff() 1736
mount() 1739
mount_parse_generic_args() 1742
mprobe() 1745
mprotect() 1748
mq_close() 1751
mq_getattr() 1753
mq_notify() 1757
mq_open() 1760
mq_receive() 1765
mq_send() 1768
mq_setattr() 1771
mq_timedreceive() 1774
mq_timedsend() 1777
mq_unlink() 1780
mrand48() 1783
_msg_info 1785
MsgDeliverEvent(), MsgDeliverEvent_r() 1788
MsgError(), MsgError_r() 1796
MsgInfo(), MsgInfo_r() 1799
MsgKeyData(), MsgKeyData_r() 1802
MsgRead(), MsgRead_r() 1810
MsgReadv(), MsgReadv_r() 1814
MsgReceive(), MsgReceive_r() 1818
MsgReceivePulse(), MsgReceivePulse_r() 1824
MsgReceivePulsev(), MsgReceivePulsev_r() 1827
MsgReceivev(), MsgReceivev_r() 1830
MsgReply(), MsgReply_r() 1834
MsgReplyv(), MsgReplyv_r() 1837
MsgSend(), MsgSend_r() 1841
MsgSendnc(), MsgSendnc_r() 1846
MsgSendPulse(), MsgSendPulse_r() 1850
MsgSendsv(), MsgSendsv_r() 1854
MsgSendsvnc(), MsgSendsvnc_r() 1858
MsgSendv(), MsgSendv_r() 1862
MsgSendvnc(), MsgSendvnc_r() 1866
MsgSendvs(), MsgSendvs_r() 1870
MsgSendvsnc(), MsgSendvsnc_r() 1874
MsgVerifyEvent(), MsgVerifyEvent_r() 1878
MsgWrite(), MsgWrite_r() 1880
MsgWritev(), MsgWritev_r() 1884
msync() 1887
munlock() 1891
munlockall() 1893
munmap() 1895
munmap_device_io() 1897
munmap_device_memory() 1899
munmap_flags() 1901
name_attach() 1904
name_close() 1911
name_detach() 1913
name_open() 1915
nanosleep() 1918
nanospin() 1920
nanospin_calibrate() 1923
nanospin_count() 1926
nanospin_ns() 1928
nanospin_ns_to_count() 1930
nap() 1933
napms() 1935
nbacnect() 1937
nbacnect_result() 1940
ND_NODE_CMP() 1942
netent 1944
netmgr_ntostr() 1945
netmgr_remote Nd() 1950
netmgr_strtond() 1952
nextafter(), nextafterf() 1954
nftw(), nftw64() 1957
nice() 1961
nrand48() 1963
nsec2timespec() 1965
ntohl() 1967
ntohs() 1969
offsetof() 1971
open(), open64() 1973
opendir() 1981
openfd() 1984
openlog() 1987
openpty() 1990
out8() 1992
out8s() 1994
out16(), outbe16(), outle16() 1996
out16s() 1998
out32(), outbe32(), outle32() 2000
out32s() 2002

C Library — P to R 2005

pathconf() 2008
pathfind(), pathfind_r() 2012
pathmgr_symmlink() 2016
pathmgr_unlink() 2018
pause() 2020
pccard_arm() 2022
pccard_attach() 2026
pccard_detach() 2028
pccard_info() 2030
pccard_lock() 2033
pccard_raw_read() 2035
pccard_unlock() 2037
pei_attach() 2039
pci_attach_device() 2041
pci_detach() 2051
pci_detach_device() 2053
pci_find_class() 2055
pci_find_device() 2057
pci_irq_routing_options() 2060
pci_map_irq() 2063
pci_present() 2065
pci_read_config() 2068
pci_read_config8() 2071
pci_read_config16() 2074
pci_read_config32() 2077
pci_rescan_bus() 2079
pci_write_config() 2081
pci_write_config8() 2084
pci_write_config16() 2086
pci_write_config32() 2088
pclose() 2090
perror() 2092
pipe() 2094
poll() 2097
popen() 2104
posix_mem_offset(), posix_mem_offset64() 2108
posix_memalign() 2111
posix_typed_mem_get_info() 2113
posix_typed_mem_open() 2116
pow(), powf() 2121
pread(), pread64() 2123
printf() 2126
procmgr_daemon() 2136
procmgr_event_notify() 2138
procmgr_event_trigger() 2143
procmgr_guardian() 2145
procmgr_session() 2149
__progname 2152
protoent 2153
pthread_abort() 2154
pthread_atfork() 2156
pthread_attr_destroy() 2158
pthread_attr_getdetachstate() 2160
pthread_attr_getguardsize() 2162
pthread_attr_getinheritsched() 2164
pthread_attr_getschedparam() 2166
pthread_attr_getschedpolicy() 2168
pthread_attr_getscope() 2170
pthread_attr_getstackaddr() 2172
pthread_attr_getstacklazy() 2174
pthread_attr_getstackprealloc() 2176
pthread_attr_getstacksize() 2178
pthread_attr_init() 2180
pthread_attr_setdetachstate() 2183
pthread_attr_setguardsize() 2185
pthread_attr_setinheritsched() 2188
pthread_attr_setschedparam() 2190
pthread_attr_setschedpolicy() 2192
pthread_attr_setscope() 2194
pthread_attr_setstackaddr() 2196
pthread_attr_setstacklazy() 2198
pthread_attr_setstackprealloc() 2200
pthread_attr_setstacksize() 2202
pthread_barrier_destroy() 2204
pthread_barrier_init() 2206
pthread_barrier_wait() 2208
pthread_barrierattr_destroy() 2210
pthread_barrierattr_getpshared() 2212
pthread_barrierattr_init() 2214
pthread_barrierattr_setpshared() 2216
pthread_cancel() 2218
pthread_cleanup_pop() 2220
pthread_cleanup_push() 2222
pthread_cond_broadcast() 2225
pthread_cond_destroy() 2227
pthread_cond_init() 2229
pthread_cond_signal() 2232
pthread_cond_timedwait() 2234
pthread_cond_wait() 2238
pthread_condattr_destroy() 2242
pthread_condattr_getclock() 2244
pthread_condattr_getpshared() 2246
pthread_condattr_init() 2248
pthread_condattr_setclock() 2250
pthread_condattr_setpshared() 2252
pthread_create() 2254
pthread_detach() 2259
pthread_equal() 2261
pthread_exit() 2263
pthread_getconcurrency() 2265
pthread_getcpuclockid() 2267
pthread_getname_np() 2269
pthread_getschedparam() 2271
pthread_getspecific() 2273
pthread_join() 2275
pthread_key_create() 2277
pthread_key_delete() 2281
pthread_kill() 2283
pthread_mutex_destroy() 2285
pthread_mutex_getprioceiling() 2287
pthread_mutex_init() 2289
pthread_mutex_lock() 2291
pthread_mutex_setprioceiling() 2295
pthread_mutex_timedlock() 2297
pthread_mutex_trylock() 2300
pthread_mutex_unlock() 2302
pthread_mutexattr_destroy() 2304
pthread_mutexattr_getprioceiling() 2306
pthread_mutexattr_getprotocol() 2308
pthread_mutexattr_getpshared() 2311
pthread_mutexattr_getrecursive() 2313
pthread_mutexattr_gettype() 2315
pthread_mutexattr_init() 2318
pthread_mutexattr_setprioceiling() 2321
pthread_mutexattr_setpshared() 2323
pthread_mutexattr_setpshared() 2325
pthread_mutexattr_setrecursive() 2327
pthread_mutexattr_settype() 2329
pthread_once() 2332
pthread_rwlock_destroy() 2335
pthread_rwlock_init() 2337
pthread_rwlock_rdlock() 2340
pthread_rwlock_timedrdlock() 2342
pthread_rwlock_timedwrlock() 2345
pthread_rwlock_tryrdlock() 2348
pthread_rwlock_trywrlock() 2350
pthread_rwlock_unlock() 2352
pthread_rwlock_wrlock() 2354
pthread_rwlockattr_destroy() 2356
pthread_rwlockattr_getpshared() 2358
pthread_rwlockattr_init() 2360
pthread_rwlockattr_setpshared() 2362
pthread_self() 2364
pthread_setscancelstate() 2365
pthread_setscancelttype() 2367
pthread_setconcurrency() 2369
pthread_setname_np() 2371
pthread_setschedparam() 2373
pthread_setspecific() 2375
pthread_sigmask() 2377
pthread_sleepon_broadcast() 2379
pthread_sleepon_lock() 2381
pthread_sleepon_signal() 2383
pthread_sleepon_timedwait() 2385
pthread_sleepon_unlock() 2389
pthread_sleepon_wait() 2391
pthread_spin_destroy() 2395
pthread_spin_init() 2397
pthread_spin_lock() 2400
pthread_spin_trylock() 2402
pthread_spin_unlock() 2404
pthread_testcancel() 2406
pthread_timedjoin() 2407

pulse 2410
pulse_attach() 2412
pulse_detach() 2416
putc() 2419
putc_unlocked() 2421
putchar() 2423
putchar_unlocked() 2425
putenv() 2427
puts() 2430
puts() 2432
puts() 2436
putw() 2439
putwc() 2441
putwchar() 2443
pwrite(), pwrite64() 2445
qnx_crypt() 2448
qsort() 2451
Raccept() 2455
raise() 2457
rand() 2460
rand_r() 2462
random() 2464
Rbind() 2467
rcmd() 2469
Rconnect() 2472
rdchk() 2474
\texttt{re\_comp()} 2476
\texttt{re\_exec()} 2478
\texttt{read()} 2480
\texttt{read\_main\_config\_file()} 2485
\texttt{readblock()} 2489
\texttt{readcond()} 2492
\texttt{readdir()} 2498
\texttt{readdir\_r()} 2502
\texttt{readlink()} 2505
\texttt{readv()} 2508
\texttt{realloc()} 2512
\texttt{realpath()} 2515
\texttt{recv()} 2517
\texttt{recvfrom()} 2520
\texttt{recvmsg()} 2524
\texttt{regcomp()} 2528
\texttt{regerror()} 2533
\texttt{regexec()} 2535
\texttt{regfree()} 2538
\texttt{remainder()}, \texttt{remainderf()} 2540
\texttt{remove()} 2542
\texttt{rename()} 2545
\texttt{res\_init()} 2548
\texttt{res\_mkquery()} 2551
\texttt{res\_query()} 2554
\texttt{res\_querydomain()} 2557
\texttt{res\_search()} 2560
\texttt{res\_send()} 2563
\texttt{resmgr\_attach()} 2566
\texttt{resmgr\_block()} 2575
\texttt{resmgr\_connect\_funcs\_t} 2578
\texttt{resmgr\_context\_alloc()} 2580
resmgr_context_free()  2583
resmgr_context_t  2585
resmgr_detach()  2587
resmgr_devino()  2591
_resmgr_handle_grow()  2594
resmgr_handle_tune()  2596
resmgr_handler()  2599
_resmgr_io_func()  2602
resmgr_io_funcs_t  2604
resmgr_iofuncs()  2609
resmgr_msgread()  2611
resmgr_msgreadv()  2613
resmgr_msgwrite()  2615
resmgr_msgwritev()  2617
_RESMGR_NPARTS()  2619
_resmgr_ocb()  2621
resmgr_open_bind()  2623
resmgr_pathname()  2626
_RESMGR_PTR()  2629
_RESMGR_STATUS()  2631
resmgr_unbind()  2633
rewind()  2635
rewinddir()  2638
Rgetsockname()  2641
rindex()  2643
rint(), rintf()  2645
Rlisten()  2648
rmdir()  2650
ROUTE  2653
Rcmd()  2660
rresvport()  2662
Rselect()  2664
rsrcregmgr_attach() 2666
rsrcregmgr_create() 2673
rsrcregmgr_destroy() 2677
rsrcregmgr_detach() 2679
rsrcregmgr_devno_attach() 2681
rsrcregmgr_devno_detach() 2685
rsrcregmgr_query() 2687
ruserok() 2690

C Library — S 2693
sbrk() 2696
scalb(), scalbf() 2699
scalbn(), scalbnf() 2702
_scalloc() 2705
scandir() 2707
scanf() 2709
sched_getparam() 2718
sched_get_priority_adjust() 2721
sched_get_priority_max() 2723
sched_get_priority_min() 2725
sched_getscheduler() 2727
sched_param 2729
sched_rr_get_interval() 2735
sched_setscheduler() 2740
sched_yield() 2743
SchedCtl(), SchedCtl_r() 2746
SchedGet(), SchedGet_r() 2781
SchedInfo(), SchedInfo_r() 2784
SchedSet(), SchedSet_r() 2787
SchedYield(), SchedYield_r() 2791
SCTP 2793
sctp_bindx() 2795
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sctp_connectx()</td>
<td>2798</td>
</tr>
<tr>
<td>sctp_freeladdr()</td>
<td>2800</td>
</tr>
<tr>
<td>sctp_freepaddr()</td>
<td>2802</td>
</tr>
<tr>
<td>sctp_getladdr()</td>
<td>2804</td>
</tr>
<tr>
<td>sctp_getpaddr()</td>
<td>2806</td>
</tr>
<tr>
<td>sctp_peeloff()</td>
<td>2808</td>
</tr>
<tr>
<td>sctp_recvmsg()</td>
<td>2810</td>
</tr>
<tr>
<td>sctp_sendmsg()</td>
<td>2813</td>
</tr>
<tr>
<td>searchenv()</td>
<td>2817</td>
</tr>
<tr>
<td>seed48()</td>
<td>2820</td>
</tr>
<tr>
<td>seekdir()</td>
<td>2822</td>
</tr>
<tr>
<td>select()</td>
<td>2824</td>
</tr>
<tr>
<td>select_attach()</td>
<td>2830</td>
</tr>
<tr>
<td>select_detach()</td>
<td>2834</td>
</tr>
<tr>
<td>select_query()</td>
<td>2837</td>
</tr>
<tr>
<td>sem_close()</td>
<td>2840</td>
</tr>
<tr>
<td>sem_destroy()</td>
<td>2842</td>
</tr>
<tr>
<td>sem_getvalue()</td>
<td>2844</td>
</tr>
<tr>
<td>sem_init()</td>
<td>2846</td>
</tr>
<tr>
<td>sem_open()</td>
<td>2849</td>
</tr>
<tr>
<td>sem_post()</td>
<td>2854</td>
</tr>
<tr>
<td>sem_timedwait()</td>
<td>2856</td>
</tr>
<tr>
<td>sem_trywait()</td>
<td>2859</td>
</tr>
<tr>
<td>sem_unlink()</td>
<td>2861</td>
</tr>
<tr>
<td>sem_wait()</td>
<td>2863</td>
</tr>
<tr>
<td>send()</td>
<td>2865</td>
</tr>
<tr>
<td>sendmsg()</td>
<td>2868</td>
</tr>
<tr>
<td>sendto()</td>
<td>2871</td>
</tr>
<tr>
<td>servant</td>
<td>2874</td>
</tr>
<tr>
<td>setbuf()</td>
<td>2875</td>
</tr>
<tr>
<td>setbuffer()</td>
<td>2877</td>
</tr>
<tr>
<td>setdomainname()</td>
<td>2880</td>
</tr>
</tbody>
</table>
setegid() 2882
setenv() 2885
seteuid() 2888
setgid() 2891
setgrent() 2894
setgroups() 2896
sethostname() 2898
sethostent() 2900
SETIOV() 2902
setitimer() 2904
setjmp() 2907
setkey() 2910
setlocale() 2912
setlogmask() 2914
setnetent() 2917
setpgid() 2921
setpgrp() 2924
setprio() 2926
setprotoent() 2928
setpwent() 2930
setregid() 2931
setreuid() 2934
setrlimit(), setrlimit64() 2937
setservent() 2944
setsid() 2946
setsockopt() 2949
setsent() 2952
setstate() 2953
settimeofday() 2955
setuid() 2957
setutent() 2960
setvbuf() 2962
_free() 2965
shm_ctl() 2967
shm_ctl_special() 2976
shm_open() 2979
shm_unlink() 2986
shutdown() 2988
sigaction() 2990
sigaddset() 2996
sigblock() 2998
sigdelset() 3000
sigemptyset() 3002
sigevent 3004
sigfillset() 3011
sigismember() 3013
siglongjmp() 3015
sigmask() 3017
signal() 3019
SignalAction(), SignalAction_r() 3023
SignalKill(), SignalKill_r() 3031
SignalProcmask(), SignalProcmask_r() 3037
SignalSuspend(), SignalSuspend_r() 3042
SignalWaitinfo(), SignalWaitinfo_r() 3045
significand(), significandf() 3048
sigpause() 3051
sigpending() 3053
sigprocmask() 3055
sigqueue() 3058
sigsetjmp() 3061
sigsetmask() 3063
sigsuspend() 3065
sigtimedwait() 3068
sigunblock() 3071
sigwait() 3073
sigwaitinfo() 3075
sin(), sinf() 3077
sinh(), sinhf() 3079
sleep() 3081
_sleepon_broadcast() 3083
_sleepon_destroy() 3085
_sleepon_init() 3087
_sleepon_lock() 3089
_sleepon_signal() 3091
_sleepon_unlock() 3093
_sleepon_wait() 3095
slogb() 3097
slogf() 3099
slogi() 3103
_smalloc() 3105
snmp_close() 3107
snmp_free_pdu() 3109
snmp_open() 3111
snmp_pdu 3113
snmp_pdu_create() 3117
snmp_read() 3120
snmp_select_info() 3122
snmp_send() 3126
snmp_session 3129
snmp_timeout() 3133
snprintf() 3135
sockatmark() 3138
socket() 3140
socketpair() 3144
SOCKSinit() 3147
sopen() 3149
sopenfd() 3154
spawn() 3157
spawnl() 3168
spawnle() 3173
spawnlp() 3178
spawnlpe() 3183
spawnp() 3188
spawnv() 3195
spawnve() 3200
spawnvp() 3205
spawnvpe() 3209
sprintf() 3214
sqrt(), sqrtf() 3217
srand() 3219
srand48() 3221
srandom() 3223
_srealloc() 3225
sscanf() 3228
stat(), stat64() 3230
statvfs(), statvfs64() 3238
stderr 3242
stdin 3243
stdout 3244
straddstr() 3245
strcasestr() 3247
strcat() 3250
strchr() 3252
strcmp() 3254
strcpi() 3256
strcoll() 3258
strncpy() 3260
strcspn() 3262
strdup() 3264
strerror() 3266
strftime() 3268
stricmp() 3274
strlen() 3276
strlwr() 3278
strncasecmp() 3280
strncat() 3283
strncmp() 3285
strncpy() 3287
strnicmp() 3290
strnset() 3292
strpbrk() 3294
strrchr() 3296
strev() 3298
strsep() 3300
strset() 3302
strsignal() 3304
strspn() 3306
strstr() 3308
strtod() 3310
strtoimax(), strtoumax() 3313
strtok() 3315
strtok_r() 3318
strtol(), strtol() 3320
strtoul(), strtoull() 3323
strupr() 3326
strxfrm() 3328
swab() 3331
swprintf() 3333
swscanf() 3335
symlink() 3337
sync() 3340
SyncCondvarSignal(), SyncCondvarSignal_r() 3341
SyncCondvarWait(), SyncCondvarWait_r() 3344
SyncCtl(), SyncCtl_r() 3349
SyncDestroy(), SyncDestroy_r() 3352
SyncMutexEvent(), SyncMutexEvent_r() 3355
SyncMutexLock(), SyncMutexLock_r() 3358
SyncMutexRevive(), SyncMutexRevive_r() 3361
SyncMutexUnlock(), SyncMutexUnlock_r() 3363
SyncSemPost(), SyncSemPost_r() 3366
SyncSemWait(), SyncSemWait_r() 3368
SyncTypeCreate(), SyncTypeCreate_r() 3371
sysconf() 3375
sysctl() 3378
syslog() 3386
sysmgr_reboot() 3389
SYSPAGE_CPU_ENTRY() 3391
SYSPAGE_ENTRY() 3393
syspage_ptr 3396
system() 3397

C Library — T to Z 3401
tan(), tanf() 3404
tanh(), tanhf() 3406
tcdrain() 3408
tcdropline() 3410
tcflow() 3413
tcflush() 3416
tcgetattr() 3419
tcgetpgrp() 3421
tcgetsid() 3423
tcgetsize() 3425
tcinj ect() 3427
tcischars() 3430
TCP 3432
tcsendbreak() 3435
tcsetattr() 3438
tcsetpgpr() 3441
tcsetsid() 3444
tcsetsize() 3446
tell(), tell64() 3448
telldir() 3451
tempnam() 3453
termios 3456
thread_pool_control() 3460
thread_pool_create() 3463
thread_pool_destroy() 3470
thread_pool_limits() 3473
thread_pool_start() 3476
ThreadCancel(), ThreadCancel_r() 3479
ThreadCreate(), ThreadCreate_r() 3483
ThreadCtl(), ThreadCtl_r() 3490
ThreadDestroy(), ThreadDestroy_r() 3498
ThreadDetach(), ThreadDetach_r() 3501
ThreadJoin(), ThreadJoin_r() 3503
time() 3506
timer_create() 3508
timer_delete() 3512
timer_getexpstatus() 3514
timer_getoverrun() 3516
timer_gettime() 3518
timer_settime() 3520
timer_timeout(), timer_timeout_r() 3523
TimerAlarm(), TimerAlarm_r() 3531
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerCreate(), TimerCreate_r()</td>
<td>3534</td>
</tr>
<tr>
<td>TimerDestroy(), TimerDestroy_r()</td>
<td>3538</td>
</tr>
<tr>
<td>TimerInfo(), TimerInfo_r()</td>
<td>3541</td>
</tr>
<tr>
<td>TimerSettime(), TimerSettime_r()</td>
<td>3545</td>
</tr>
<tr>
<td>TimerTimeout(), TimerTimeout_r()</td>
<td>3549</td>
</tr>
<tr>
<td>times()</td>
<td>3557</td>
</tr>
<tr>
<td>timespec</td>
<td>3560</td>
</tr>
<tr>
<td>timespec2nsec()</td>
<td>3561</td>
</tr>
<tr>
<td>timezone</td>
<td>3563</td>
</tr>
<tr>
<td>tm</td>
<td>3564</td>
</tr>
<tr>
<td>tmpfile(), tmpfile64()</td>
<td>3566</td>
</tr>
<tr>
<td>tmpnam()</td>
<td>3569</td>
</tr>
<tr>
<td>tolower()</td>
<td>3572</td>
</tr>
<tr>
<td>toupper()</td>
<td>3574</td>
</tr>
<tr>
<td>towctrans()</td>
<td>3576</td>
</tr>
<tr>
<td>towlower()</td>
<td>3578</td>
</tr>
<tr>
<td>towupper()</td>
<td>3580</td>
</tr>
<tr>
<td>TraceEvent()</td>
<td>3582</td>
</tr>
<tr>
<td>truncate()</td>
<td>3585</td>
</tr>
<tr>
<td>ttymame()</td>
<td>3588</td>
</tr>
<tr>
<td>ttymame_r()</td>
<td>3590</td>
</tr>
<tr>
<td>tzname</td>
<td>3592</td>
</tr>
<tr>
<td>tzset()</td>
<td>3593</td>
</tr>
<tr>
<td>ualarm()</td>
<td>3596</td>
</tr>
<tr>
<td>UDP</td>
<td>3599</td>
</tr>
<tr>
<td>ultoa(), ulltoa()</td>
<td>3601</td>
</tr>
<tr>
<td>umask()</td>
<td>3604</td>
</tr>
<tr>
<td>umount()</td>
<td>3607</td>
</tr>
<tr>
<td>UNALIGNED_PUT16()</td>
<td>3609</td>
</tr>
<tr>
<td>UNALIGNED_PUT32()</td>
<td>3611</td>
</tr>
<tr>
<td>UNALIGNED_PUT64()</td>
<td>3613</td>
</tr>
<tr>
<td>UNALIGNED_RET16()</td>
<td>3615</td>
</tr>
</tbody>
</table>
UNALIGNED RET32() 3617
UNALIGNED RET64() 3619
uname() 3621
ungetc() 3624
ungetwc() 3626
UNIX 3628
unlink() 3631
unsetenv() 3634
usleep() 3636
utime() 3638
utimes() 3641
utmp 3644
utmpname() 3646
utoa() 3648
va_arg() 3651
va_copy() 3657
va_end() 3659
va_start() 3661
valloc() 3663
verr(), verrx() 3665
vfork() 3667
vfprintf() 3669
vfscanf() 3672
vfwprintf() 3675
vfscanf() 3677
vprintf() 3679
vscanf() 3681
vslogf() 3684
vsnprintf() 3686
vsscanf() 3689
vsscanf() 3692
vswprintf() 3695
vswscanf() 3697
vsyslog() 3699
vwarn(), vwarnx() 3701
vwprintf() 3703
vwscanf() 3705
wait() 3707
wait3() 3710
wait4() 3713
waitid() 3717
waitpid() 3720
warn(), warnx() 3723
wctomb() 3725
wcschr() 3729
wcscmp() 3731
wcscoll() 3733
wcscpy() 3735
wcscspn() 3737
wcsftime() 3739
wcslen() 3741
wcscat() 3743
wcscmp() 3745
wcscpy() 3747
wcspbrk() 3749
wcschr() 3751
wcsrtombs() 3753
wcsstr() 3755
wcstol(), wcstof(), wcstold() 3759
wcstol(), wcstoul(), wcstoul() 3760
wcstok() 3764
wcstol(), wcstoull() 3766
SOCKS — A Basic Firewall 3821

A

About SOCKS 3823
How to SOCKSify a client 3823
What SOCKS expects 3824

B

Third-Party Copyright Notices 3827

BSD stack 3829
BSD stack and various utilities 3830
MINIX operating system 3837
ncurses library 3838
C Summary of Safety Information 3843

Cancellation points 3845
Interrupt handlers 3850
Signal handlers 3853
Multithreaded programs 3867

Glossary 3871

Index 3895
List of Figures

A hierarchy of processes. 1790
A deadlock when sending messages improperly among processes. 1790

`MsgSendv()`, client to process manager. 1803
`MsgReplyv()`, process manager to client. 1803
`MsgSendv()`, client to filesystem manager 1804
Components of a fully qualified pathname. 1946
Specifying a guardian for child processes. 2145
Conditions that satisfy an input request. 2493
Most of the `spawn*()` functions do a lot of work before a message is sent to `procnto`. 3159
About This Reference
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:

<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 format single-step instructions like this:

➤ To reload the current page, press `Ctrl – R`.

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.
What you’ll find in this guide

The Library Reference describes the C functions, data types, and protocols that are included as part of the QNX Neutrino RTOS.

The Library Reference also contains:

- Summary listings of the library, including a description of what you’ll find in a function description
- Summary of safety information:
  - functions that are cancellation points
  - functions that you can safely call from an interrupt handler
  - functions that you can safely call from a signal handler
  - functions that you can’t safely call from a multithreaded program.
- descriptions of manifests
- SOCKS — A Basic Firewall
- Third-Party Copyright Notices
- Glossary

What’s new in QNX Neutrino 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.
What's new in QNX Neutrino 6.3.2

 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
What’s new in QNX Neutrino 6.3.2

- RLIMIT_NPROC
- RLIMIT_NTHR
- RLIMIT_OFILE
- RLIMIT_RSS

`iowfunc_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
What's new in QNX Neutrino 6.3.0 Service Pack 2

New Content

```c
cache_fini()
```
Free resource when the driver is unloaded.

```c
CACHE_FLUSH()
```
Flush cache line associated to a data buffer.

```c
cache_init()
```
Register with the cache coherency library.

```c
CACHE_INVALID()
```
Invalidate cache line associated to a data buffer.

Changed Content

```c
mq_close(), mq_getattr(), mq_notify(), mq_open(), mq_receive(),
mq_send(), mq_setattr(), mq_timedreceive(), mq_timedsend(),
mq_unlink()
```
Added information about the traditional (`mqueue`) and alternate (`mq`) implementations of message queues.

```c
nanospin(), nanospin_calibrate(), nanospin_count(), nanospin_ns(),
nanospin_ns_to_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.

```c
pci_attach_device(), pci_find_class(), pci_find_device()
```

For a list of supported device and vendor IDs, see `<hw/pci_devices.h>`; for a list of class and subclass codes, see `<hw/pci.h>`.
Errata

```c
_cmdname()
```
Corrected the information about what this function returns.

```c
openlog()
```
Corrected the name of the LOG_FTP facility.

```c
pci_attach_device()
```
The `BusNumber` and `DevFunc` members of the `pci_dev_info` structure are input/output.

```c
pci_irq_routing_options()
```
This function is for x86 only.

```c
sem_close(), sem_open(), sem_unlink()
```
Named semaphores are now managed by `procnto`, not `mqueue`.

```c
thread_pool_create()
```
The description of the `block_func` member of the `thread_pool_attr_t` structure has been corrected.

```c
vfprintf(), vprintf()
```
The code samples now compile without warnings.

What’s new in QNX Neutrino 6.3.0 Service Pack 1

New content

```c
asyncmsg_channel_create()
```
Create an asynchronous message channel.

```c
asyncmsg_channel_destroy()
```
Destroy an asynchronous message channel.

```c
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.

shmCtl_special()
    Give special attributes to a shared memory object

**Changed content**

getdomainname()
    If the buffer isn’t large enough, getdomainname() truncates the domain name.

getgrouplist() 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*() functions use negative \textit{errno} values to indicate failure, so you shouldn't pass a negative value for the \textit{status} to \textit{MsgReply*()}, because the \textit{MsgSend*} functions could interpret it as an error code.

\textit{name_attach}()

The example now handles an \texttt{IO\_CONNECT} message.

nanospin(), nanospin_ns(), and nanospin_ns_to_count()

The first time that you call these functions, the C library invokes \texttt{nanospin\_calibrate}() with an argument of 0 (interrupts enabled), unless you call it directly first.

\textit{pci\_attach\_device}()

Added PCI\_MASTER\_ENABLE to the flags.

\textit{procmgr\_daemon}()

The data in the \texttt{siginfo\_t} structure for the \texttt{SIGCHLD} signal that the parent receives isn't useful in this case.

\textit{rsrcdbmgr\_create}()

Added RSRCDBMGR\_FLAG\_NOREMOVE to the flags.

\textit{setenv}()

This function doesn't free any memory. If you want to change the value of an existing environment variable, you should use \textit{putenv}() instead.

\textit{sigevent}

If you don’t want to modify the priority of the thread that receives the pulse, specify \texttt{SIGEV\_PULSE\_PRIO\_INHERIT} for the \textit{priority} when you call \texttt{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 × 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 × 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(), spawnp(), spawnpv() 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()`.

`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()`.
What's new in QNX Neutrino 6.2.1

New content

`sctp_freepaddrs()`
Free all resources allocated by `sctp_getpaddrs()`.

`sctp_getladdr()` Return all locally bound addresses on a socket.

`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_recvmsg()` Receive message using advanced SCTP features.

`sctp_sendmsg()` Send message using advanced SCTP features.

`tmpfile64()` Large-file support for `tmpfile()`.

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.

`errno` 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.
Changed content

*bind*, *bindresvport*

These functions aren’t cancellation points any more, because this conflicted with POSIX.

*htonl*, *htons*, *inet_ntop*, *inet_pton*, *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.

*execle*, *execve*

You can now execute a shell script.

*fgetc*, *fgetchar*, *fgets*, *fgetwc*, *fgetws*, *getc*, *gete unlocked*,

*getchar*, *getchar unlocked*, *getws*, *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.
shm_open() Corrected the interpretation of the name argument.

sigaction() Corrected the example (it isn’t safe to call printf() in a signal handler).

spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe() You can now execute a shell script.
The child process’s tms_utime, tms_stime, tms_cutime, and tms_cstime are now calculated.

timer_create() Don’t use SIGEV_INTR or SIGEV_UNBLOCK for the event type.

vsnprintf() Corrected the returned values.

What’s new in QNX Neutrino 6.2

Significant changes:

- New content
- Deprecated content
- Errata

New Content

addinfo TCP/IP address information
dircntl() Control an open directory
freeaddrinfo() Free an address information structure
freeifaddrs() Free an address information structure
gai_strerror() Return the getaddrinfo() error code
getaddrinfo() Get address information
getdomainname() Get the domain name of the current host
What’s new in QNX Neutrino 6.2

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_nton() 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_iофuncs() 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
sleepon_destroy() Destroy a sleepon
sleepon_init() Initialize a sleepon
sleepon_lock() Lock a sleepon
What's new in the QNX Neutrino 6.1.0 docs © 2006, QNX Software Systems GmbH & Co. KG.

_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  

wcsstoimax(), 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
- `SyncMutexRevive()`, `SyncMutexRevive_r()`
  Revive a mutex
- `thread_pool_control()`
  Control the thread pool behavior
What's new in the QNX Neutrino 6.1.0 docs © 2006, QNX Software Systems GmbH & Co. KG.

thread_pool_limits()
   Wrapper function for thread_pool_control()

TraceEvent()     Trace kernel events

**Deprecated content**

matherr()        Handle errors in math library functions
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. `fopen()`, `fread()`, `fwrite()`, and `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 (0D) 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.
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
Summary of function categories

*aio_return()*  Get the return status for an asynchronous I/O operation
*aio_suspend()*  Wait for asynchronous I/O operations to complete
*aio_write()*   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
Summary of function categories

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
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_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
strtoimax(), strtoumax()

Convert a string into an integer

strtol(), strtohl()

Convert a string into a long integer

strtoul(), strtohll()

Convert a string into an unsigned long integer

timespec

Time-specification structure

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
- dircntl() 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
Summary of function categories

`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
- `pulseDetach()`: Detach a handler function from a pulse code
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resmgr_attach()</td>
<td>Attach a path to a pathname space</td>
</tr>
<tr>
<td>resmgr_block()</td>
<td>Block while waiting for a message</td>
</tr>
<tr>
<td>resmgr_connect_funcs_t</td>
<td>Table of POSIX-level connect functions</td>
</tr>
<tr>
<td>resmgr_context_alloc()</td>
<td>Allocate a resource-manager context</td>
</tr>
<tr>
<td>resmgr_context_free()</td>
<td>Free a resource-manager context</td>
</tr>
<tr>
<td>resmgr_context_t</td>
<td>Context information that’s passed between</td>
</tr>
<tr>
<td></td>
<td>resource-manager functions</td>
</tr>
<tr>
<td>resmgr_detach()</td>
<td>Remove a pathname from the pathname space</td>
</tr>
<tr>
<td>resmgr_devino()</td>
<td>Get the device and inode number</td>
</tr>
<tr>
<td>resmgr_handle_grow()</td>
<td>Expand the capacity of the device manager</td>
</tr>
<tr>
<td></td>
<td>database</td>
</tr>
<tr>
<td>resmgr_handle_tune()</td>
<td>Tune aspects of client fd-to-OCB mapping</td>
</tr>
<tr>
<td>resmgr_handler()</td>
<td>Handle resource manager messages</td>
</tr>
<tr>
<td>_resmgr_io_func()</td>
<td>Retrieve an I/O function from an I/O function table</td>
</tr>
<tr>
<td>resmgr_iofuncs()</td>
<td>Extract the I/O function pointers associated with</td>
</tr>
<tr>
<td></td>
<td>client connections</td>
</tr>
</tbody>
</table>
**Summary of function categories**

QNX Software Systems GmbH & Co. KG.

*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()`
Assocrate 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)

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
Summary of function categories

- `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

- `ltruncate()`  
  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
Summary of function categories

- **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
- **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
Summary of function categories

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

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
**Summary of function categories**

- `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.
- `pthread_mutex_setprioceiling()`: Set a mutex’s priority ceiling.
- `pthread_mutex_timedlock()`: Lock a mutex.
- `pthread_mutex_trylock()`: Attempt to lock a mutex.
Summary of Functions

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

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
**Summary of function categories**

- `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
- `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
Summary of Functions

- **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
- **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
Summary of function categories

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sem_trywait()</code></td>
<td>Wait on a semaphore, but don’t block</td>
</tr>
<tr>
<td><code>sem_unlink()</code></td>
<td>Destroy a named semaphore</td>
</tr>
<tr>
<td><code>sem_wait()</code></td>
<td>Wait on a semaphore</td>
</tr>
<tr>
<td><code>sync()</code></td>
<td>Synchronize filesystem updates</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pci_attach()</code></td>
<td>Connect to the PCI server</td>
</tr>
<tr>
<td><code>pci_attach_device()</code></td>
<td>Attach a driver to a PCI device</td>
</tr>
<tr>
<td><code>pci_detach()</code></td>
<td>Disconnect from the PCI server</td>
</tr>
<tr>
<td><code>pci_detach_device()</code></td>
<td>Detach a driver from a PCI device</td>
</tr>
<tr>
<td><code>pci_find_class()</code></td>
<td>Find devices that have a specific Class Code</td>
</tr>
<tr>
<td><code>pci_find_device()</code></td>
<td>Find the PCI device with a given device ID and vendor ID</td>
</tr>
<tr>
<td><code>pci_irq_routing_options()</code></td>
<td>Retrieve PCI IRQ routing information</td>
</tr>
<tr>
<td><code>pci_map_irq()</code></td>
<td>Map an interrupt pin to an IRQ</td>
</tr>
<tr>
<td><code>pci_present()</code></td>
<td>Determine whether or not PCI BIOS is present</td>
</tr>
<tr>
<td><code>pci_read_config()</code></td>
<td>Read from the configuration space of a PCI device</td>
</tr>
</tbody>
</table>
Summary of function categories

\[\text{pci\_read\_config8()}\]
Read a byte from the configuration space of a device

\[\text{pci\_read\_config16()}\]
Read 16-bit values from the configuration space of a device

\[\text{pci\_read\_config32()}\]
Read 32-bit values from the configuration space of a device

\[\text{pci\_rescan\_bus()}\]
Rescan the PCI bus for added or removed devices

\[\text{pci\_write\_config()}\]
Write to the configuration space of a PCI device

\[\text{pci\_write\_config8()}\]
Write bytes to the configuration space of a PCI device

\[\text{pci\_write\_config16()}\]
Write 16-bit values to the configuration space of a device

\[\text{pci\_write\_config32()}\]
Write 32-bit values to the configuration space of a device

\[\text{hwi\_find\_item()}\]
Find an item in the \text{hwi\_item} structure

\[\text{hwi\_find\_tag()}\]
Find a tag in the \text{hwi\_item} structure

\[\text{hwi\_off2tag()}\]
Return a pointer to the start of a tag in the \text{hwinfo} area of the system page

\[\text{hwi\_tag2off()}\]
Return the offset from the start of the \text{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

Absolute values

- `abs()` Return the absolute value of an integer
- `cabs()`, `cabsf()` Compute the absolute value of a complex number
- `fabs()`, `fabsf()` Compute the absolute value of a double number
- `labs()` Calculate the absolute value of a long integer

Bessel functions

- `j0()`, `j0f()` Compute a Bessel function of the first kind
- `j1()`, `j1f()` Compute a Bessel function of the first kind
- `jn()`, `jn()`, `jn()` Compute a Bessel function of the first kind
- `y0()`, `y0f()` Compute a Bessel function of the second kind
Summary of function categories

*y1(), y1f()  Compute a Bessel function of the second kind
*yn(), ynf()  Compute a Bessel function of the second kind

**Division, remainders, and modular arithmetic**

*div()  Calculate the quotient and remainder of a division operation
*drem(), dremf()  Compute the remainder of two numbers
*fmod(), fmodf()  Compute a residue, using floating-point modular arithmetic
*ldiv()  Perform division on long integers
*modf(), modff()  Break a number into integral and fractional parts
*remainder(), remainderf()  Compute the floating point remainder

**Floating-point settings**

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

*fp_exception_mask()  Get or set the current exception mask
*fp_exception_value()  Get the value of the current exception registers
*fp_precision()  Set or get the current precision
*fp_rounding()  Set or get the current rounding
**Gamma functions**

- `gamma()`, `gamma_r()`, `gammaf()`, `gammaf_r()`
  - Log gamma function

- `lgamma()`, `lgamma_r()`, `lgammaf()`, `lgammaf_r()`
  - Log gamma function

**Logarithms and exponentials**

The following routines calculate logarithms and exponentials:

- `exp()`, `expf()`  
  - Compute the exponential function of a number

- `expm1()`, `expm1f()`  
  - Compute the exponential of a number, then subtract 1

- `frexp()`, `frexpf()`  
  - Break a floating-point number into a normalized fraction and an integral power of 2

- `ilogb()`, `ilogbf()`  
  - Compute the integral part of a logarithm

- `ldexp()`, `ldexpf()`  
  - Multiply a floating-point number by an integral power of 2

- `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
Summary of function categories

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

Miscellaneous

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

erf(), erf()  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(), isninf()  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
lrand48() Generate a pseudo-random nonnegative long integer
mrand48() Generate a pseudo-random signed long integer
nrand48() Generate a pseudo-random nonnegative long integer in a thread-safe manner
seed48() Initialize a sequence of pseudo-random numbers
srand48() Initialize a sequence of pseudo-random numbers
Summary of function categories

Roots and powers

- `cbrt()`, `cbrtf()` Compute the cube root of a number
- `pow()`, `powf()` Raise a number to a given power
- `sqrt()`, `sqrtf()` Calculate the nonnegative square root of a number

Rounding

- `ceil()`, `ceilf()` Round up a value to the next integer
- `floor()`, `floorf()` Round down a value to the next integer
- `rint()`, `rintf()` Round to the nearest integral value

Trigonometric and hyperbolic functions

- `acos()`, `acosf()` Compute the arccosine of an angle
- `acosh()`, `acoshf()` Compute the inverse hyperbolic cosine
- `asin()`, `asinf()` Compute the arcsine of an angle
- `asinh()`, `asinhf()` Compute the inverse hyperbolic sine
- `atan()`, `atanf()` Compute the arctangent of an angle
- `atanh()`, `atanhf()` Compute the inverse hyperbolic tangent
- `atan2()`, `atan2f()` Compute the arctangent, determining the quadrant
- `cos()`, `cosf()` Compute the cosine of an angle
- `cosh()`, `coshf()` Compute the hyperbolic cosine
- `sin()`, `sinf()` Calculate the sine of an angle
- `sinh()`, `sinhf()` Compute the hyperbolic sine
- `tan()`, `tanf()` Calculate the tangent of an angle
- `tanh()`, `tanhf()` Calculate the hyperbolic tangent
Memory allocation functions

These functions allocate and deallocate blocks of memory:

- **alloca()** - Allocate automatic space from the stack
- **_amblksiz** - The increment for the break pointer
- **_btext** - The beginning of the text segment
- **calloc()** - Allocate space for an array
- **_edata** - The end of the data segment, excluding BSS data
- **_end** - The end of the data segment, including BSS data
- **_etext** - The end of the text segment
- **free()** - Dealloacate a block of memory
- **ftw()** - Walk a file tree
- **longjmp()** - Restore the environment saved by **setjmp()**
- **mallinfo()** - Get memory allocation information
- **malloc()** - Allocate memory
- **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
- **realloc()** - Allocate, reallocate or free a block of memory
Summary of function categories

© 2006, QNX Software Systems GmbH & Co. KG.

Summary of functions

$sbrk()$ 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

$ffs()$ 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
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
### Summary of functions

- **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

*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
Summary of function categories

- **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

- **MsgSendvnc(), MsgSendvnc 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)

- **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 recommend 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
- `_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
- `fdataasync()` 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
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
Summary of function categories

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

readcond()  
Read data from a terminal device

readlink()  
Place the contents of a symbolic link into a buffer

readv()  
Read bytes from a file

select()  
Check for files that are ready for reading or writing

SETIOV()  
Fill in the fields of an iov_t structure

sopen()  
Open a file for shared access

sopenfd()  
Open for shared access a file associated with a given descriptor

symlink()  
Create a symbolic link to a path

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

tell(), tell64()  
Determine the current file position
umask() Set the file mode creation mask for the process
uname() Get information about the operating system
unlink() Delete a file
write() Write bytes to a file
writeblock() Write blocks of data to a file
writev() Write bytes to a file

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
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:
Summary of function categories

`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

`intr_v86()`  
Execute a real-mode software interrupt

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

- **argv**
  - A pointer to the vector of arguments passed to `main()`

- **auxv**
  - A pointer to a vector of auxiliary arguments to `main()`

- **clearenv()**
  - Clear the process environment area

- **confstr()**
  - Get configuration-defined string values

- **ctermid()**
  - 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
getsid() Get the session ID of a process
getuid() Get the user ID
getutent() Read the next entry from the user-information file
getutid() Search for an entry in the user-information file
getutline() Get an entry from the user-information file
initgroups() Initialize the supplementary group access list
isatty() Test to see if a file descriptor is associated with a terminal
login_tty() Prepare for a login in a tty
main() The function where program execution begins
ND_NODE_CMP() Compare two node descriptor IDs
netmgr_ntostr() Convert a node descriptor into a string
**Summary of Function Categories**

- **netmgr_remote_nd()**
  - Get a node descriptor that’s relative to a remote node

- **netmgr_strtond()**
  - Convert a string into a node descriptor

- **_progname**
  - The basename of the program being executed

- **putenv()**
  - Add, change, or delete an environment variable

- **pututline()**
  - Write an entry in the user-information file

- **searchenv()**
  - Search the directories listed in an environment variable

- **setegid()**
  - Set the effective group ID for a process

- **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
Summary of function categories

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)

vwarn(), vwarnx()  
Formatted error message (varargs)

warn(), warnx()  
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 Jr()**
Establish a connection between a process and a channel

**ConnectClientInfo(), ConnectClientInfo Jr()**
Store information about a client connection

**ConnectDetach(), ConnectDetach Jr()**
Break a connection between a process and a channel

**ConnectFlags(), ConnectFlags Jr()**
Modify the flags associated with a connection

**ConnectServerInfo(), ConnectServerInfo Jr()**
Store information about a connection

**daemon()**
Run a program in the background

**DebugBreak()**
Enter the process debugger

**DebugKDBreak()**
Enter the kernel debugger

**DebugKDOutput()**
Print text with the kernel debugger

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

**dladdr()**
Translate an address to symbolic information
Summary of function categories

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dlclose()</code></td>
<td>Close a shared object</td>
</tr>
<tr>
<td><code>dlerror()</code></td>
<td>Get dynamic loading diagnostic information</td>
</tr>
<tr>
<td><code>dlopen()</code></td>
<td>Gain access to an executable object file</td>
</tr>
<tr>
<td><code>dlsym()</code></td>
<td>Get the address of a symbol in a shared object</td>
</tr>
<tr>
<td><code>execl()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execl()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execlp()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execlpe()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execv()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execve()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execvp()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>execvpe()</code></td>
<td>Execute a file</td>
</tr>
<tr>
<td><code>_exit()</code></td>
<td>Terminate the program</td>
</tr>
<tr>
<td><code>exit()</code></td>
<td>Terminate the program</td>
</tr>
<tr>
<td><code>fork()</code></td>
<td>Create a new process</td>
</tr>
<tr>
<td><code>forkpty()</code></td>
<td>Create a new process operating in a pseudo-tty</td>
</tr>
<tr>
<td><code>getrlimit()</code>, <code>getrlimit64()</code></td>
<td>Get the limit on a system resource</td>
</tr>
<tr>
<td><code>getprio()</code></td>
<td>Get the priority of a given process</td>
</tr>
<tr>
<td><code>InterruptAttach()</code>, <code>InterruptAttachEvent()</code></td>
<td>Attach an interrupt handler to an interrupt source</td>
</tr>
<tr>
<td><code>InterruptAttachEvent()</code></td>
<td>Attach an event to an interrupt source</td>
</tr>
</tbody>
</table>
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

SchedCtl()  
Control the adaptive partitioning scheduler

SchedGet(), SchedGet.r()  
Get the scheduling policy for a thread

SchedInfo(), SchedInfo.r()  
Get scheduler information

SchedSet(), SchedSet.r()  
Set the scheduling policy for a thread

SchedYield(), SchedYield.r()  
Yield to other threads

setitimer()  
Set the value of an interval timer

setprio()  
Set the priority of a process

setrlimit(), setrlimit64()  
Set the limit on a system resource
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sigaction()</code></td>
<td>Examine or specify the action associated with a signal</td>
</tr>
<tr>
<td><code>sigaddset()</code></td>
<td>Add a signal to a set</td>
</tr>
<tr>
<td><code>sigblock()</code></td>
<td>Add to the mask of signals to block</td>
</tr>
<tr>
<td><code>sigdelset()</code></td>
<td>Delete a signal from a set</td>
</tr>
<tr>
<td><code>sigemptyset()</code></td>
<td>Initialize a set to contain no signals</td>
</tr>
<tr>
<td><code>sigfillset()</code></td>
<td>Initialize a set to contain all signals</td>
</tr>
<tr>
<td><code>sigismember()</code></td>
<td>See if a given signal is in a given set</td>
</tr>
<tr>
<td><code>signmask()</code></td>
<td>Construct a mask for a signal number</td>
</tr>
<tr>
<td><code>signal()</code></td>
<td>Set handling for exceptional conditions</td>
</tr>
<tr>
<td><code>SignalAction()</code></td>
<td>Examine and/or specify actions for signals</td>
</tr>
<tr>
<td><code>SignalKill()</code></td>
<td>Send a signal to a process group, process or thread</td>
</tr>
<tr>
<td><code>SignalProcmask()</code></td>
<td>Modify or examine the signal blocked mask of a thread</td>
</tr>
<tr>
<td><code>SignalSuspend()</code></td>
<td>Suspend a process until a signal is received</td>
</tr>
<tr>
<td><code>SignalWaitinfo()</code></td>
<td>Select a pending signal</td>
</tr>
<tr>
<td><code>sigpause()</code></td>
<td>Wait for a signal</td>
</tr>
<tr>
<td><code>sigpending()</code></td>
<td>Examine the set of pending, masked signals for a process</td>
</tr>
<tr>
<td><code>sigprocmask()</code></td>
<td>Examine or change the signal mask for a process</td>
</tr>
</tbody>
</table>
Summary of function categories

- **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
- **sigunblock()**: 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
- **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
**Summary of function categories**

- \textit{SyncCtl()}, \textit{SyncCtl.r()}
  
  Perform an operation on a synchronization object

- \textit{SyncDestroy()}, \textit{SyncDestroy.r()}
  
  Destroy a synchronization object

- \textit{SyncMutexEvent()}, \textit{SyncMutexEvent.r()}
  
  Attach an event to a mutex

- \textit{SyncMutexLock()}, \textit{SyncMutexLock.r()}
  
  Lock a mutex synchronization object

- \textit{SyncMutexUnlock()}, \textit{SyncMutexUnlock.r()}
  
  Unlock a mutex synchronization object

- \textit{SyncMutexRevive()}, \textit{SyncMutexRevive.r()}
  
  Revive a mutex that’s in the DEAD state

- \textit{SyncSemPost()}, \textit{SyncSemPost.r()}
  
  Increment a semaphore

- \textit{SyncSemWait()}, \textit{SyncSemWait.r()}
  
  Wait on a semaphore

- \textit{system()}
  
  Execute a system command

- \textit{SYSPAGE\_CPU\_ENTRY()}
  
  Return a CPU-specific entry from the system page

- \textit{SYSPAGE\_ENTRY()}
  
  Return an entry from the system page

- \textit{syspage_ptr}
  
  A pointer to the system page

- \textit{ThreadCancel()}, \textit{ThreadCancel.r()}
  
  Cancel a thread

- \textit{ThreadCreate()}, \textit{ThreadCreate.r()}
  
  Create a new thread
ThreadCtl(), ThreadCtl rhetorical
Control a thread

ThreadDestroy(), ThreadDestroy rhetorical
Destroy a thread immediately

ThreadDetach(), ThreadDetach rhetorical
Detach a thread from a process

ThreadJoin(), ThreadJoin rhetorical
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
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_getres()` Get the resolution of the clock
- `clock_gettime()` Get the current time of a clock
- `clock_nanosleep()` High resolution sleep with specified clock
- `clock_settime()` Set a clock
- `getitimer()` 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()`
- `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

sched_yield() Yield to other READY processes at the same priority

timer_create() Create a timer

timer_delete() Delete a timer

timer_getexpstatus()
Get the expiry status of a timer

timer_getoverrun()
Return the number of timer overruns

timer_gettime() Get the amount of time left on a timer

timer_settime() Set the expiration time for a timer
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_fstype_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

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

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
Summary of function categories

**iofunc_lock_calloc()**
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

**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

iofunc_read_default()  
Default handler for _IO_READ messages

iofunc_readlink()  
Verify a client’s ability to read a symbolic link

iofunc_read_verify()  
Verify a client’s read access to a resource

iofunc_rename()  
Do permission checks for a  
_IO_CONNECT_RENAME message

iofunc_space_verify()  
Do permission checks for _IO_SPACE message

iofunc_stat()  
Populate a stat structure
Summary of function categories

`iofunc_stat_default()`  
Default handler for `IO_STAT` messages

`iofunc_sync()`  
Indicate if synchronization is needed

`iofunc_sync_default()`  
Default handler for `IO_SYNC` messages

`iofunc_sync_verify()`  
Verify permissions to sync

`iofunc_time_update()`  
Update time stamps

`iofunc_unblock()`  
Unblock OCBs

`iofunc_unblock_default()`  
Default unblock handler

`iofunc_unlink()`  
Verify that an entry can be unlinked

`iofunc_unlock_ocb_default()`  
Default handler for the `unlock_ocb` callout

`iofunc_utime()`  
Update time stamps

`iofunc_utime_default()`  
Default handler for `IO_UTIME` messages

`iofunc_write_default()`  
Default handler for `IO_WRITE` messages

`iofunc_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

rsrcregmgr_attach()  
Reserve a system resource for a process

rsrcregmgr_create()  
Create a system resource

rsrcregmgr_destroy()  
Destroy a system resource

rsrcregmgr_detach()  
Return a system resource to the resource database

rsrcregmgr_devno_attach()  
Get a major and minor number

rsrcregmgr_devno_detach()  
Detach a major and minor number

rsrcregmgr_query()  
Query the resource database

umount()  
Unmount a filesystem

**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.):

alphasort()  
Compare two directory entries
Summary of function categories

*bsearch()* Perform a binary search on a sorted array

*ffs()* Find the first bit set in a bit string

*hcreate()* Create a hash search table

*hdestroy()* Destroy the hash search table

*hsearch()* Search the hash search table

*index()* Find a character in a string

*lfind()* Find entry in a linear search table

*lsearch()* Linear search and update

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

*qsort()* Sort an array, using a modified Quicksort algorithm

*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

*rindex()* Find a character in a string

*strcspn()* Count the characters at the beginning of a string that aren’t in a character set

*strstr()* Find one string inside another
Shared memory functions

These functions provide memory mapping capabilities:

- `mmap()`, `mmap64()`  
  Map a memory region into a process address space
- `mprotect()`  
  Change memory protection
- `munmap()`  
  Unmap previously mapped addresses
- `shm_ctl()`  
  Give special attributes to a shared memory object
- `shm_open()`  
  Open a shared memory object
- `shm_unlink()`  
  Remove a shared memory object

Signal functions

These functions deal with handling and sending signals.

- `DebugBreak()`  
  Enter the process debugger
- `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
### Summary of Functions

- **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

#### 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
Summary of function categories

**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.)

- `clearerr()`  
  Clear the end-of-file and error indicators for a stream
- `fclose()`  
  Close a stream
- `fcloseall()`  
  Close all open stream files, except `stdin`, `stdout` and `stderr`
Summary of function categories

- `fdopen()`    Associate a stream with a file descriptor
- `feof()`     Test the end-of-file indicator
- `ferror()`   Test the error indicator for a stream
- `fflush()`   Flush the input or output buffer for a stream
- `fgetc()`    Get the next character from a file stream
- `fgetchar()` Get a character from `stdin`
- `fgetpos()`  Store the current position of a stream
- `fgets()`    Get a string of characters from a stream
- `flushall()` Clear all input buffers and write all output buffers
- `fopen()`    Open a stream
- `fprintf()`  Write output to a stream
- `fputc()`    Write a character to an output stream
- `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
Summary of function categories

Summary of Functions November 2, 2006

Summary of function categories

© 2006, QNX Software Systems GmbH & Co. KG.

`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
Summary of function categories

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localeconv()</td>
<td>Set numeric formatting according to the current locale</td>
</tr>
<tr>
<td>re_comp()</td>
<td>Compile a regular expression</td>
</tr>
<tr>
<td>re_exec()</td>
<td>Execute a regular expression</td>
</tr>
<tr>
<td>regcomp()</td>
<td>Compile a regular expression</td>
</tr>
<tr>
<td>regerror()</td>
<td>Explain a regular expression error code</td>
</tr>
<tr>
<td>regexec()</td>
<td>Compare a string with a compiled regular expression</td>
</tr>
<tr>
<td>regfree()</td>
<td>Release memory allocated for a regular expression</td>
</tr>
<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>strcsnp()</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>
</tbody>
</table>
Summary of function categories

strerror() Map an error number to an error message
stricmp() Compare two strings, ignoring case
strlen() Compute the length of a string
strlwr() Convert a string to lowercase
strncasecmp() Compare two strings, ignoring case, up to a given length
strncat() Concatenate two strings, up to a maximum length
strncmp() Compare two strings, up to a given length
strncpy() Copy a string, to a maximum length
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)
Summary of function categories

strupr() Convert a string to uppercase
strxfrm() Transform one string into another, to a given length
vscanf() Write formatted output to a buffer (varargs)
vsprintf() 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
- “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
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_expnd() 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
Summary of function categories

- `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
- `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

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
Summary of functions

`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_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_nameindex()`  
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
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inet_pton()</code></td>
<td>Convert a text host address to a numeric network address</td>
</tr>
<tr>
<td>INET6</td>
<td>Internet Protocol version 6 family</td>
</tr>
<tr>
<td><code>inet6_option_*()</code></td>
<td>Manipulate IPv6 hop-by-hop and destination options</td>
</tr>
<tr>
<td><code>inet6_rthdr_*()</code></td>
<td>Manipulate IPv6 Router header options</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPsec</td>
<td>Internet security protocol</td>
</tr>
<tr>
<td><code>ipsec_dump_policy()</code></td>
<td>Generate readable string from IPsec policy specification</td>
</tr>
<tr>
<td><code>ipsec_get_policylen()</code></td>
<td>Get length of the IPsec policy</td>
</tr>
<tr>
<td><code>ipsec_set_policy()</code></td>
<td>Generate IPsec policy specification structure from readable string</td>
</tr>
<tr>
<td><code>ipsec_strerror()</code></td>
<td>Error code for IPsec policy manipulation library</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol version 6</td>
</tr>
<tr>
<td><code>isfdtype()</code></td>
<td>Determine whether a file descriptor refers to a socket</td>
</tr>
<tr>
<td><code>listen()</code></td>
<td>Listen for connections on a socket</td>
</tr>
<tr>
<td><code>nbaconnect()</code></td>
<td>Initiate a connection on a socket (nonblocking)</td>
</tr>
<tr>
<td><code>nbaconnect_result()</code></td>
<td>Get the status of the previous call to <code>nbaconnect()</code></td>
</tr>
<tr>
<td><code>netent</code></td>
<td>Structure for information from the network database</td>
</tr>
<tr>
<td><code>ntohl()</code></td>
<td>Convert network-byte order value</td>
</tr>
</tbody>
</table>
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)

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

Rgethostname() Get the name of a socket (via a SOCKS server)

Rlisten() Listen for connections on a socket (via a SOCKS server)
### Summary of function categories

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROUTE</strong></td>
<td>System packet forwarding database</td>
</tr>
<tr>
<td><em>Rcmd()</em></td>
<td>Execute a command on a remote host (via a SOCKS server)</td>
</tr>
<tr>
<td><em>rresvport()</em></td>
<td>Obtain a socket with a privileged address</td>
</tr>
<tr>
<td><em>Rselect()</em></td>
<td>Check for descriptors that are ready for reading or writing (via a SOCKS server)</td>
</tr>
<tr>
<td><em>ruserok()</em></td>
<td>Check the identity of a remote host</td>
</tr>
<tr>
<td><strong>SCTP</strong></td>
<td>Stream Control Transmission Protocol</td>
</tr>
<tr>
<td><em>sctp_bindx()</em></td>
<td>Add or remove one or more given addresses from an association</td>
</tr>
<tr>
<td><em>sctp_connectx()</em></td>
<td>Connect a host to a multihomed endpoint</td>
</tr>
<tr>
<td><em>sctp_freeladdr()</em></td>
<td>Free all resources allocated by <em>sctp_getladdr()</em></td>
</tr>
<tr>
<td><em>sctp_freepaddr()</em></td>
<td>Free all resources allocated by <em>sctp_getpaddr()</em></td>
</tr>
<tr>
<td><em>sctp_getladdr()</em></td>
<td>Get all locally bound addresses on a socket</td>
</tr>
<tr>
<td><em>sctp_getpaddr()</em></td>
<td>Get all peer addresses in an association</td>
</tr>
<tr>
<td><em>sctp_peeloff()</em></td>
<td>Branch off an association into a separate socket</td>
</tr>
<tr>
<td><em>sctp_recvmsg()</em></td>
<td>Receive a message, using advanced SCTP features</td>
</tr>
<tr>
<td><em>sctp_sendmsg()</em></td>
<td>Send a message, using advanced SCTP features</td>
</tr>
<tr>
<td><em>send()</em></td>
<td>Send a message to a socket</td>
</tr>
<tr>
<td><em>sendmsg()</em></td>
<td>Send a message to a socket</td>
</tr>
<tr>
<td><em>sendto()</em></td>
<td>Send a message to a socket</td>
</tr>
<tr>
<td><strong>servent</strong></td>
<td>Structure for information from the services database</td>
</tr>
<tr>
<td><em>setdomainname()</em></td>
<td>Set the domain name of the current host</td>
</tr>
</tbody>
</table>
Summary of function categories
© 2006, QNX Software Systems GmbH & Co. KG.

sethostent() Set the local hosts entry
sethostname() Set the name of the current host
setnetent() Open the network database
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
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
Summary of function categories

**tcgetsid()**
Get the process group ID of the session leader for a controlling terminal

**tcgetsize()**
Get the size of a character device

**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

**tcsetattr()**
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_abrupt()**
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

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
Summary of function categories

`pthread_attr_setschedpolicy()`  
Set the thread scheduling policy attribute

`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_getcpu_clockid()  
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
Summary of function categories

- `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

- `pthread_mutex_destroy()`  
  Destroy a mutex
Summary of functions

- `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_sleepon_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

- `pthread_rwlockattr_init()`: Create a read-write lock attribute object
Summary of function categories

`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

`pthread_setcancelstate()`  
Set a thread’s cancellation state
**Summary of function categories**

*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()*

Unlock waiting threads

*pthread_sleepon_lock()*

Unlock 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

*pthread_spin_lock()*

Lock a thread spinlock
Summary of function categories

pthread_spin_trylock()  
Try to lock a thread spinlock

pthread_spin_unlock()  
Unlock a thread spinlock

pthread_testcancel()  
Test the thread cancellation

_sleepon_broadcast()  
Wake up multiple threads

_sleepon_destroy()  
Destroy a sleepon lock

_sleepon_init()  
Initialize a sleepon lock

_sleepon_lock()  
Lock a sleepon lock

_sleepon_signal()  
Wake up a single thread

_sleepon_unlock()  
Unlock a sleepon lock

_sleepon_wait()  
Wait on a sleepon 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

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_gettime()    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

diff_time()    Calculate the difference between two times
ftime()    Get the current time, and store it in a structure

gmtime()    Convert calendar time to a broken-down 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

mtime()    Convert local time to calendar time

settimeofday()    Set the time and date
Summary of function categories

- `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 <code>stdin</code></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>
### Summary of Function Categories

- **swscanf()**: Scan input from a wide character string
- **towctrans()**: Convert a wide character in a specified manner
- **tolower()**: Convert a wide character to lowercase
- **toupper()**: Convert a wide character to uppercase
- **ungetwc()**: Push a wide character back onto an input stream
- **vfwscanf()**: Scan input from a file (varargs)
- **vfwprintf()**: Write formatted wide-character output to a file (varargs)
- **vfwscanf()**: Scan input from a file (varargs)
- **vswprintf()**: Write formatted wide-character output to a buffer (varargs)
- **vswscanf()**: Scan input from a string (varargs)
- **vwprintf()**: Write formatted wide-character output to standard output (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
- **wcsnoll()**: Compare two wide-character strings, using the locale’s collating sequence
- **wcscpy()**: Copy a wide-character string
- **wcsespn()**: 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
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)
wcsrchr() Find the last occurrence of a wide character in a string
wcsspn() Count the wide characters at the beginning of a string that are in a given character set
wcsstr() Find one wide-character string inside another
wcstod(), wcstof(), wcstold() Convert a wide-character string into a double, float, or long double
wcstoimax(), wcstoumax() 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
wcstombs() 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

```
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:

```
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**

  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.
What's in a function description?

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.
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 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
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><strong>BEGIN_DECLS</strong></td>
<td>sys/platform.h</td>
<td>Denotes start of C code for a C++ compiled program.</td>
</tr>
<tr>
<td><strong>BIG_ENDIAN</strong></td>
<td>sys/platform.h</td>
<td>Code is compiled for a big-endian target.</td>
</tr>
<tr>
<td><strong>CHAR_SIGNED</strong></td>
<td>sys/platform.h</td>
<td>Code is compiled with the char type defaulting to <code>signed</code>.</td>
</tr>
<tr>
<td><strong>CHAR_UNSIGNED</strong></td>
<td>sys/platform.h</td>
<td>Code is compiled with the char type defaulting to <code>unsigned</code>.</td>
</tr>
<tr>
<td><strong>END_DECLS</strong></td>
<td>sys/platform.h</td>
<td>Denotes end of C code for a C++ compiled program.</td>
</tr>
<tr>
<td><strong>INT_BITS</strong></td>
<td>sys/platform.h</td>
<td>The number of bits in the <code>int</code> datatype.</td>
</tr>
<tr>
<td><strong>LITTLE_ENDIAN</strong></td>
<td>sys/platform.h</td>
<td>Code is compiled for a little-endian target.</td>
</tr>
<tr>
<td><strong>LONG_BITS</strong></td>
<td>sys/platform.h</td>
<td>The number of bits in the <code>long</code> datatype.</td>
</tr>
<tr>
<td><strong>NTO_VERSION</strong></td>
<td>sys/neutrino.h</td>
<td>A version number times 100 (e.g. 2.00 is 200).</td>
</tr>
<tr>
<td><strong>PTR_BITS</strong></td>
<td>sys/platform.h</td>
<td>The number of bits in a void pointer.</td>
</tr>
<tr>
<td><strong>OPTIMIZE</strong></td>
<td>sys/platform.h</td>
<td>Code is compiled for optimization.</td>
</tr>
<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>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Manifest</th>
<th>Header file to include</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
**abort()**

*Raise the SIGABRT signal to terminate program execution*

**Synopsis:**

```c
#include <stdlib.h>

void abort( void );
```

**Library:**

```c
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:**

```c
#include <stdlib.h>

int main( void )
{
    int major_error = 1;
    if( major_error )
        abort();

    /* You’ll never get here. */
    return EXIT_SUCCESS;
}
```
**abort()**

© 2006, 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>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()`, `exec()`, `execle()`, `execlp()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `.exit()`, `exit()`, `getenv()`, `main()`, `putenv()`, `sigaction()`, `signal()`, `spawn*()` functions, `system()`, `wait()`, `waitpid()`
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
```
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:

cabs(), fabs(), labs()
accept()

Accept a connection on a socket

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:

```c
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

<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:

```
bind(), close(), connect(), listen(), select(), socket()
```
access() — 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.
  - `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 `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.
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.
EINVVAL   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 access() 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.

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

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(), eaccess, errno, fstat(), open(), stat()
acos(), acosf()  
Compute the arccosine of an angle

Synopsis:

```c
#include <math.h>

double acos( double x );

float acosf( 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 arccos (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:

```c
#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>
<th></th>
</tr>
</thead>
<tbody>
<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:**

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

```
#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:

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

int main( void )
{
    printf( "%f\n", acosh( 1.5 ) );

    return EXIT_SUCCESS;
}
```

produces the output:
acosh(), acoshf()

0.962424

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`. 
addrinfo

\( ai\_addr \) Binary socket address.

\( ai\_next \) A pointer to the next \texttt{addrinfo} structure in the linked list.

\textbf{Classification:}

POSIX 1003.1

\textbf{See also:}

\texttt{freeaddrinfo()}, \texttt{gai_strerror()}, \texttt{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()

<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>
**Synopsis:**

```c
#include <aio.h>

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

**Library:**

```c
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 `aiocbp` 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>
Asynchronously synchronize a file

**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()

**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>
Asynchronously read from a file

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…
```
### Safety

<table>
<thead>
<tr>
<th>Thread</th>
<th>Yes</th>
</tr>
</thead>
</table>

*QNX Neutrino Functions and Macros*  
November 2, 2006
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

<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>
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:

```c
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>Yes/No</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

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…
## Safety

<table>
<thead>
<tr>
<th>Thread</th>
<th>Yes</th>
</tr>
</thead>
</table>

*aio_write()*
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 \( \text{(unsigned)}-1 \) is returned (\( \text{errno} \) 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;
}
```
**alarm()**

**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:

```
#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:

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>

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

/* allocate temporary buffer for file name */
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

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>
alloca()  

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:

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

\footnote{This function is in \texttt{libc.a}, but not in \texttt{libc.so} (in order to save space).}

Description:

The \texttt{alphasort()} function alphabetically compares two directory entries. You can use it as the \texttt{compar} argument to \texttt{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.
alphasort()

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(), rewinddir(), scandir(), seekdir(), telldir()
**Synopsis:**

```c
#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.*
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:

`__argv, __auxv, getopt(), main()`
A pointer to the vector of arguments passed to 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()`
**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
```

November 2, 2006  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>
</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>

**asctime_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>
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( "%.16f\n", asinh( 0.5 ) );
    return EXIT_SUCCESS;
}
```

produces the output:

`0.481212`
asinh(), asinhf()

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:

acosh(), atanh(), sinh(), errno
assert()  
© 2006, QNX Software Systems GmbH & Co. KG.
Print a diagnostic message and optionally terminate the program

Synopsis:

```
#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:

```
assert((fd = open("filename", O_RDONLY)) != -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
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() © 2006, QNX Software Systems GmbH & Co. KG.

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
asynmsg_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:**

- `asynmsg_channel_destroy()`, `asynmsg_connect_attach()`,
- `asynmsg_connect_attr()`, `asynmsg_connect_detach()`,
- `asynmsg_flush()`, `asynmsg_free()`, `asynmsg_get()`,
- `asynmsg_malloc()`, `asynmsg_put()`, `asynmsg_putv()`

*ChannelCreate()*

*Asynchronous Messaging Technote*
asyncmsg_channel_destroy()

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

continued…

174 QNX Neutrino Functions and Macros

November 2, 2006
asyncmsg_channel_destroy()

<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:

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_connect_attach() © 2006, QNX Software Systems GmbH & Co. KG.

Establish a connection between a process and a channel

Synopsis:

```c
#include <sys/neutrino.h>
#include <sys/asyncmsg.h>

int asyncmsg_connect_attach;
    (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.
**Library:**

```c
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>

November 2, 2006   QNX Neutrino Functions and Macros  177
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()

Asynchronous Messaging Technote
asyncmsg_connect_attr()

Return the original connection attributes

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_connect_attr(), 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
asyncmsg_connect_detach()

Break a connection between a process and a channel

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() © 2006, QNX Software Systems GmbH & Co. KG.

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:

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:

```
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:

- \texttt{*call\_back} Callback function for notification.
- \texttt{err} The error status of the package.
- \texttt{buff} A pointer to the package buffer.
- \texttt{handle} A handle associated with the callback function.
- \texttt{buffer\_size} The message buffer size.
- \texttt{max\_num\_buffer} The maximum number of buffers allowed for this connection.
- \texttt{trigger\_num\_msg} Uses the number of the pending message as triggering criteria.
- \texttt{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
asyncmsg_flush()

<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_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>

November 2, 2006 QNX Neutrino Functions and Macros 187
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()

Asynchronous Messaging Technote
asyncmsg_malloc()

Allocate a message buffer for sending

Synopsis:

```c
#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

<table>
<thead>
<tr>
<th>Safety</th>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>continued…</td>
<td></td>
</tr>
</tbody>
</table>

November 2, 2006 QNX Neutrino Functions and Macros 191
asyncmsg_malloc()

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_flush(), asyncmsg_free(),  
asyncmsg_get(), asyncmsg_put(), asyncmsg_putv()  

Asynchronous Messaging Technote
asyncmsg_put(),
asyncmsg_putv()

Send an asynchronous message to a connection

Synopsis:
#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_connect attach() will be used.
**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

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

*continued…*
Asynchronous Messaging

Technote

November 2, 2006

QNX Neutrino Functions and Macros

195
**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 (-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:**

```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()
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 (-π, 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.
**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`
Compute an inverse hyperbolic tangent

**Synopsis:**

```c
#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:**

```c
#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

**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:**

`acosh(), asinh(), errno, tanh()`
atexit()

Register functions to be called during normal program termination

Synopsis:

```c
#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:
  ```c
  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()`.

Note:

The functions registered with `atexit()` aren’t called when the program terminates with a call to `_exit()`. 
atexit()

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
atexit()

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(), _exit(), exit()
Synopsis:

```
#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 `strtof()` like this:

```
strtof( 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:

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

int main( void )
{
    double x;
```
atof()

```c
x = atof( "3.1415926" );
printf( "x = %f\n", x );
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`, `sscanf()`, `strtod()`
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;
}
```
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:

```c
#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:

```c
#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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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:**

`atol()`, `itoa()`, `ltoa()`, `sscanf()`, `strtol()`, `strtoul()`, `ultoa()`, `utoa()`
Synopsis:

```c
#include <stdlib.h>

long atol( const char* ptr );
long long atoll( const char* nptr );
```

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( "x = %ld\n", x );
    return EXIT_SUCCESS;
}
```
produces the output:

\[ x = -289 \]

**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:**

atoi(), itoa(), ltoa(), sscanf(), strtol(), strtoul(), ultoa(), utoa()
atomic_add()
Safely add to a variable

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:

```c
#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.

Note: 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:

```
#include <atomic.h>
...
volatile unsigned count;
unsigned previous;
...
previous = atomic_add_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_clr(), atomic_clr_value(), atomic_set(),
atomic_set_value(), atomic_sub(), atomic_sub_value(),
atomic_toggle(), atomic_toggle_value()
atomic_clr()

Safely clear a variable

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

<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_set(), atomic_set_value(), atomic_sub(), atomic_sub_value(), atomic_toggle(), atomic_toggle_value()
Synopsis:

#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:

```c
#include <atomic.h>
...
volatile unsigned flags;
unsigned previous;
...
previous = atomic_clr_value( &flags, 0x10101010 );
```

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_set(),
atomic_set_value(), atomic_sub(), atomic_sub_value(),
atomic_toggle(), atomic_toggle_value()
atom ic_set()  

Safely set bits in a variable

Synopsis:

```c
#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:

```c
libc
```

Use the `-l c` 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:

```c
#include <atomic.h>
...
```
### atomic_set()

```c
volatile unsigned flags;
...
atomic_set( &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_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:

```c
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:

#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:

#include <atomic.h>

...
volatile unsigned count;

atomic_sub( &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_add_value(), atomic_clr(), atomic_clr_value(),
atomic_set(), atomic_set_value(), atomic_sub_value(), atomic_toggle(),
atomic_toggle_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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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>
<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_toggle()`, `atomic_toggle_value()`
atomic_toggle()

Synopsis:

```c
#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:

```c
#include <atomic.h>

...
volatile unsigned flags;
...

atomic_toggle( &flags, 0xdeadbeef );

**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_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.
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()
A pointer to a vector of auxiliary arguments to main()

**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()`
**basename()**

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

### Synopsis:

```c
#include <libgen.h>

char* basename( 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 `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>&quot;/usr/lib&quot;</td>
<td>&quot;lib&quot;</td>
</tr>
<tr>
<td>&quot;/usr/&quot;</td>
<td>&quot;usr&quot;</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
</tbody>
</table>

**Classification:**

POSIX 1003.1 XSI
basename()

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:

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.
bcmp()

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

<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:

`bcopy(), bzero(), memcmp(), strcmp()`
bcopy()

Copy a number of characters in one string to another

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 )
```

QNX Neutrino Functions and Macros November 2, 2006
{  
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

<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:

`bcmp()`, `bzero()`, `memmove()`, `strcpy()`
bind()  
© 2006, QNX Software Systems GmbH & Co. KG.

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

<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:

- ICMP, IP, TCP, and UDP protocols
- `connect()`, `getsockname()`, `listen()`, `socket()`
bindresvport()
Bind a socket to a privileged IP port

Synopsis:

```
#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.

EADDRNOTAVAIL The specified address isn’t available from the local machine.

E_BADF Invalid descriptor sd.

EFAULT The sin parameter isn’t a valid pointer to a sockaddr_in structure.

E_INVAL 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:

connect(), getsockname(), listen(), socket()
Synopsis:

```
#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>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
Safety

<table>
<thead>
<tr>
<th>Signal handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<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`, `execl()`, `execle()`, `execlp()`, `execlepe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `free()`, `getrlimit()`, `malloc()`, `mmap()`, `sbrk()`
Perform a binary search on a sorted array

**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()

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_KEYWORDS sizeof(keywords) / sizeof(char *)

int kw_compare( const void *p1, const void *p2 )
{
    const char *p1c = (const char *) p1;
    const char **p2c = (const char **) p2;
    return( strcmp( p1c, *p2c ) );
}

int keyword_lookup( const char *name )
{
    const char **key;

    key = (char const **) bsearch( name, keywords, NUM_KEYWORDS, sizeof( char * ), kw_compare );
    if( key == NULL ) return( -1 );
}
```


```c
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**

<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:**

`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()
btowc()

Convert a single-byte character to a wide character

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

- 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:

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**bzero()**

Set the first part of an object to null bytes

**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:**

*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

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:

`bcmp()`, `bcopy()`, `memset()`, `strset()`
Compute the absolute value of a complex number

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 ) );
```
cabs(), cabsf()

**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**

<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()`, `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:

- `libcachef`

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

<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>
</tbody>
</table>

continued…
### cache_fini()

| Safety | Thread | Yes |

**See also:**

`CACHE_FLUSH()`, `cache_init()`, `CACHE_INVAL()`
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 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 \textit{cache\_init()}.  

\textbf{Environment Variables}  

The following environment variables, if they exist, affect the behavior of this cache coherency function:

\begin{itemize}
  \item \textbf{CACHE\_NOP} Instructs the library that the \textit{CACHE\_FLUSH()} and \textit{CACHE\_INVAL()} macros should have no effect.
  \item \textbf{CACHE\_MSYNC} Instructs the library that the \textit{CACHE\_FLUSH()} and \textit{CACHE\_INVAL()} macros should use the \textit{msync()} \texttt{C} library call to perform cache synchronization.
\end{itemize}

\textit{This function was added in the QNX Momentics 6.3.0 SP2.}

\textbf{Classification:}

\begin{itemize}
  \item QNX Neutrino
\end{itemize}

\textbf{Safety}

\begin{tabular}{ll}
  Cancellation point & No \\
  Interrupt handler & Yes \\
  Signal handler & Yes \\
  Thread & Yes \\
\end{tabular}

\textbf{Caveats:}

\textit{CACHE\_FLUSH()} is implemented as a macro.
See also:

`cache_fini()`, `cache_init()`, `CACHE_INVAL()`
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.
**cache_init()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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.

\textit{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 \textit{cache\_flush\_rate} field.

\textit{fd}  The driver should set this field to NOFD.

Other fields in the structure should not be referenced or modified.

\textbf{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:

QNX Neutrino

### Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

### 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_init()`, `CACHE_FLUSH()`, `CACHE_INVAL()`
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()`. 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 pre-determine 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 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 be not be corrupted by the

November 2, 2006 QNX Neutrino Functions and Macros 271
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_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.

For more information about cache coherency, see `cache_init()`.

This function was added in the QNX Momentics 6.3.0 SP2.

**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:

 CACHE_INVAL() is implemented as a macro.

See also:

cache_fini(), CACHE_FLUSH(), 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.

![Note: 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.
**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()`
**Synopsis:**

```c
#include <math.h>

double cbrt ( double x );
float cbrtf ( 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 `cbrt()` and `cbrtf()` functions compute the cube root of `x`.

**Returns:**

The cube root of `x`. If `x` is `NAN`, `cbrt()` 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 = cbrt(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

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()
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 ≥ `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 ),
```
```c
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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, cfgetospeed(), cfsetispeed(), cfsetospeed(), tcgetattr(),
tcsetattr(), termios
cfgetospeed()  
Return the output baud rate that’s stored in a termios structure

Synopsis:

```c
#include <termios.h>

speed_t cfgetospeed(
    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 `cfgetospeed()` function returns the output 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 output 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 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(), cfsetspeed(), tcgetattr(),
tcsetattr(), termios
Synopsis:

```c
#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 **libc.a** but not in **libc.so** (in order to save space).

**Description:**

The `cfgopen()` function opens the configuration file named by `path`. This function is a cover function for `open()` that searches several default system locations for your files, based on specified characteristics.

The value of `flags` correspond to, and have similar limitations of, the standard `open()` flags. The `flags` value is constructed by the bitwise ORing of values from the following list, defined in the `<cfgopen.h>` header file. Applications must specify exactly one of these file-access modes in the value of `flag`:

- `CFGFILE_RDONLY`  
  Open for reading only.
- `CFGFILE_RDWR`  
  Open for reading and writing.
- `CFGFILE_WRONLY`  
  Open for writing only.

You can also include any combination of these bits in the value of `flag`:

- `CFGFILE_APPEND`  
  If set, the file offset is set to the end of the file prior to each write.
- `CFGFILE_CREAT`  
  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 `chmod()`).
CFGFILE_EXCL

If CFGFILE_EXCL and CFGFILE_CREAT are set, and the file exists, \texttt{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) \textit{path}, there are several bits that you can specify and OR into \textit{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 \textit{node\_name} is the value you get by calling \texttt{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>

**See also:**

`confstr()`, `fcfgopen()`, `open()`

`mib.txt`, `snmpd.conf` in the *Utilities Reference*
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 &= ~(ECHO|ECHONL|ICANON|ISIG|IEXTEN);
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:

- `0` Success.
- `-1` An error occurred (`errno` indicates the reason).
**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:**

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>

QNX Neutrino Functions and Macros November 2, 2006
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()`
**cfsetispeed()**

Set the input baud rate in a `termios` structure

**Synopsis:**

```c
#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:**

```c
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 `tgetattr()`.
cfsetispeed()

- 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
    */
```

November 2, 2006  QNX Neutrino Functions and Macros 295
```
    /*
    speed = 9600;
    cfsetispeed( &termios_p, speed );
    tcsetattr( fd, TCSADRAIN, &termios_p);
    
    close( fd );
    return EXIT_SUCCESS;
  }
```

**Classification:**

POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**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()`.

- 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.

- 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.
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
**cfsetospeed()**

<table>
<thead>
<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`, `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `tcgetattr()`, `tcsetattr()`, `termios`
**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:**

- `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.
_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_CODETHREADDEATH |
| 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_CODEUNBLOCK |
| 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 \textit{MsgSendv()}. Setting the \_NTO\_CHF\_UNBLOCK bit in \textit{flags} prevents a thread that’s in the \textsc{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 \textit{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:

\textit{ChannelCreate()}  

The channel ID of the newly created channel. If an error occurs, the function returns -1 and sets \textit{errno}.  

\textit{ChannelCreate\_r()}
ChannelCreate(), ChannelCreate_r() © 2006, 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:**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>All kernel channel objects are in use.</td>
</tr>
<tr>
<td>EBUSY</td>
<td>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.</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:**

ChannelDestroy(), close(), ConnectAttach(), ConnectDetach(), _msg_info, MsgReceivev(), MsgReplyv(), MsgSendv(), MsgSendPulse(), name_close(), _pulse
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.
ChannelDestroy(), ChannelDestroy_r()

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:

EINVAL 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()
Chdir()  Change the current working directory

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.
- `ENAMEETOOLONG`: 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.
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.
chmod()

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

<table>
<thead>
<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(), 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 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.
- **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 <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

<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()`, `errno`, `fchown()`, `fstat()`, `lchown()`, `open()`, `stat()`
**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:**

- `EACCES`  
  Search permission is denied for a component of `path`.

- `EBADF`  
  The descriptor isn’t valid.

- `EFAULT`  
  The `path` argument points to an illegal address.
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

<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:

`chdir()`
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.

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**

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**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).
clearenv()

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

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:

The `clearenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

`environ`, `errno`, `execl()`, `execlp()`, `execlpe()`, `execv()`, `execve()`,
`execvp()`, `execvpe()`, `getenv()`, `putenv()`, `searchenv()`, `setenv()`, `spawn()`,
`spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnvp()`,
`spawnve()`, `spawnvp()`, `spawnvpe()`, `system()`, `unsetenv()`
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 */
        }
    }
}
```
clearerr()

```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>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:**

`feof()`, `ferror()`, `fopen()`, `perror()`, `rewind()`
clock()

Return the number of clock ticks used by the program

Synopsis:

```
#include <time.h>

clock_t clock( void );
```

Library:

```
libc
```

Use the `-l c` 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.

Examples:

```
#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++ ) {
```
```c
for( j = 1; j <= 100; j++ ) {
    x += sqrt( (double) i * j );
}

printf( "%.16f\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**

<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(), difftime(), gmtime(), localtime(),
localtime_r(), mktime(), strftime(), time(), tzset()`
clock_getcpuclockid()

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.

Errors:

ESRCH No process can be found corresponding to the specified pid.

Classification:

POSIX 1003.1 CPT

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:

clock_gettime(), clock_gettime(), ClockId(), clock_settime(),
pthread_getcpuclockid(), timer_create()
clock_getres()

Get the resolution of the clock

Synopsis:

```c
#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`.

November 2, 2006 QNX Neutrino Functions and Macros 335
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>
</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_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.
EINVAL     Invalid clock id.
ESRCH      The process associated with this request doesn’t exist.

Examples:

```c
/*
 * 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()

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
**Synopsis:**

```c
#include <time.h>

int clock_nanosleep( clockid_t clock_id, 
                   int flags, 
                   const struct timespec * rqtpt, 
                   struct timespec * rmtpt);
```

**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 `rqtpt` 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 `rqtpt` argument (TIMER_ABSTIME is set).

  If, at the time of the call, the time value specified by `rqtpt` is less than or equal to the time value of the
specified clock, then \texttt{clock.nanosleep()} returns immediately, and the calling process isn’t suspended.

- 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 \texttt{clock.nanosleep()} with \texttt{TIMER_ABSTIME} \textit{not set}, and \texttt{clock.id} set to \texttt{CLOCK_REALTIME} is the equivalent to calling \texttt{nanosleep()} with the same \texttt{rqtp} and \texttt{rmtpp} arguments.

\texttt{rqtp} A pointer to a \texttt{timespec} structure that specifies the time interval between the requested time and the time actually slept.

\texttt{rmtpp} 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 \texttt{clock.nanosleep()} function, if \texttt{rmtpp} 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 \texttt{clock.nanosleep()} function has no effect on the structure referenced by \texttt{rmtpp}.

\textbf{Library:}

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

\textbf{Description:}

The \texttt{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 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

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:

clock_settime(), nanosleep(), sleep(), timespec
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 \texttt{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" );
    }

    return 0;
}
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>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:**

`clock_getres`, `clock_gettime`, `errno`, `timespec`
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 \textit{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 \textit{ClockAdjust()} and \textit{ClockAdjust()}_\textit{r()} functions are identical except in the way they indicate errors. See the Returns section for details.

The total time adjustment, in nanoseconds, is:

\[
(new->\text{tick\_count} \times new->\text{tick\_nsec\_inc})
\]

If the current clock is ahead of the desired time, you can specify a negative \textit{tick\_nsec\_inc} to slow down the clock. This is preferable to setting the time backwards with the \textit{ClockTime()} kernel call, since some programs may malfunction if time goes backwards.

Picking small values for \textit{tick\_nsec\_inc} and large values for \textit{tick\_count} adjusts the time slowly, while the opposite approach adjusts it rapidly. As a rule of thumb, don’t try to set a \textit{tick\_nsec\_inc} that exceeds the basic clock tick as set by the \textit{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 \textit{tick\_count} and \textit{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:

\textit{ClockAdjust()} \quad If an error occurs, the function returns -1 and sets \textit{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.
EINVAL 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ClockPeriod(), ClockTime()
ClockCycles() © 2006, QNX Software Systems GmbH & Co. KG.

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 \texttt{ClockCycles()} on an SMP machine, you must use the following call to “lock” the thread to a single CPU:

\texttt{ThreadCtl(_NTO_TCTL_RUNMASK, \ldots)}

\subsection*{Blocking states:}
This call doesn’t block.

\section*{CAUTION:}
Be careful about wrapping of the cycle counter. Use the following to calculate how many seconds before the cycle counter wraps:

\begin{equation}
\neg\text{(uint64}_t)0 /\text{SYSPAGE\_ENTRY(qtime)} \rightarrow \text{cycles\_per\_sec}
\end{equation}

\subsection*{Examples:}
See \texttt{SYSPAGE\_ENTRY()}.  

\subsection*{Classification:}
QNX Neutrino

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & \text{No} \\
Interrupt handler & \text{No} \\
Signal handler & \text{Yes} \\
Thread & \text{Yes} \\
\hline
\end{tabular}
\end{table}
See also:

SYSPAGE_ENTRY(), ThreadCtl()
**ClockId(), ClockId_r()**

Get a clock ID for a given process and thread

**Synopsis:**
```
#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_gettime()` 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 = %.2f\n", (1000000000.0 - (stop-start)) / 10000000.0);
}
```
ClockId(), ClockId_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>

See also:

ClockTime(), clock_gettime(), pthread_gettime()
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 TDK 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 EINV AL. 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 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
ClockPeriod(), ClockPeriod_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>

See also:

`clock_getres()`, `ClockAdjust()`
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
\texttt{clock_gettime()} or \texttt{clock_settime()}.

If \texttt{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 \texttt{id} is \texttt{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 \texttt{ClockPeriod()} kernel call.

You need to have superuser privileges to set the clock.

\textbf{Blocking states}

These calls don’t block.

\textbf{Returns:}

The only difference between these functions is the way they indicate
errors:

\texttt{ClockTime()} If an error occurs, the function returns -1 and sets
\texttt{errno}. Any other value returned indicates success.

\texttt{ClockTime_r()} EOK is returned on success. This function does
\textbf{NOT} set \texttt{errno}. If an error occurs, the function
returns a value in the Errors section.

\textbf{Errors:}

\texttt{EFAULT} A fault occurred when the kernel tried to access the
buffers provided.
ClockTime(), ClockTime_r() © 2006, QNX Software Systems GmbH & Co. KG.

EINVVAL 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

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(), clock_settime(), ClockAdjust(), ClockPeriod()
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`.
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>
</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:

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
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`, `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>
</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:

- `openlog()`, `setlogmask()`, `syslog()`, `vsyslog()`
- `logger`, `syslogd` in the *Utilities Reference*
Synopsis:

```
#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

<table>
<thead>
<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:

`cmdname()`, `progname`
cmdname()

Find the path used to invoke the current process

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>
```
```c
#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**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
See also:

`basename()`, `cmdfd()`, `progname`
Confstr()

Get configuration-defined string values

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).
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 \leq 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()  © 2006, QNX Software Systems GmbH & Co. KG.
Initiate a connection on a socket

Synopsis:

```c
define <sys/types.h>
define <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()

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.

The socket is already connected.

The network isn’t reachable from this host.

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**

<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

`accept()`, `bind()`, `getsockname()`, `nbacconnect()`, `select()`, `socket()`
ConnectAttach(),
ConnectAttach_r()

Establish a connection between a process and a channel

Synopsis:

```
#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_SID_CHANNEL into 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_SID_CHANNEL (e.g. during open()), however, it’s unlikely that any applications would want to call it this way.

flags If flags contains _NTO_COF_CLOEXEC, the connection is closed when your process calls an exec*() function to start a new process.

Library:

libc

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

Description:

The ConnectAttach() and ConnectAttach_r() kernel calls establish a connection between the calling process and the channel specified by chid owned by the process specified by pid on the node specified by 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 int representing the connection. The system returns the first available connection ID starting at the value specified by the index argument. Any thread in the calling process can use either MsgSendv() to send messages or MsgSendPulse() to send pulses over the connection. The connection ID is used directly as a POSIX file descriptor (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 \texttt{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 \texttt{nd} objects</td>
<td>\texttt{ND_NODE_CMP()}</td>
</tr>
<tr>
<td>Convert a \texttt{nd} to text</td>
<td>\texttt{netmgr_ndtostr()}</td>
</tr>
<tr>
<td>Convert text to a \texttt{nd}</td>
<td>\texttt{netmgr_strtond()}</td>
</tr>
</tbody>
</table>

**Returns:**

The only difference between these functions is the way they indicate errors:

\textit{ConnectAttach()} — A connection ID that’s used by the message primitives. If an error occurs, the function returns -1 and sets \texttt{errno}.

\textit{ConnectAttach\_r()} — A connection ID that’s used by the message primitives. This function does NOT set \texttt{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:

QNX Neutrino

\begin{tabular}{l l}
\textbf{Safety} & \\
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}

See also:

\textit{ChannelCreate()}, \textit{ConnectDetach()}, \textit{execl()}, \textit{execl()}, \textit{execvp()},
\textit{execle()}, \textit{execv()}, \textit{execve()}, \textit{execvp()}, \textit{MsgSendPulse()},
\textit{MsgSendv()}, \textit{netmgr\_remote\_nd()}

November 2, 2006
ConnectClientInfo(), ConnectClientInfo_r()

Store information about a client connection

Synopsis:

```c
#include <sys/neutrino.h>

int ConnectClientInfo( int scoid,
                       struct clientinfo * info
                       int ngroups );

int ConnectClientInfo_r( int scoid,
                        struct clientinfo * 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.
ConnectClientInfo(),
ConnectClientInfo_r()

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.

To: Use this function:

<table>
<thead>
<tr>
<th>Compare two nd objects</th>
<th>ND_NODE_CMP()</th>
</tr>
</thead>
<tbody>
<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>
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.

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:

ConnectServerInfo(), _msg_info, MsgInfo(), MsgReceivev(), ND_NODE_CMP(), netmgr_ntostr(), netmgr_remote_nd(), netmgr_strond()
ConnectDetach(), ConnectDetach_r()

Break a connection between a process and a channel

Synopsis:

```
#include <sys/neutrino.h>

int ConnectDetach( int coid );
int ConnectDetach_r( int coid );
```

Arguments:

- \textit{coid}  
  The connection ID of the connection you want to break.

Library:

\textit{libc}

Use the \texttt{-l c} option to qcc to link against this library. This library is usually included automatically.

Description:

The \textit{ConnectDetach()} and \textit{ConnectDetach_r()} kernel calls detach the connection specified by the \textit{coid} argument. If any threads are blocked on the connection (\textit{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:

\textit{ConnectDetach()}

If an error occurs, the function returns -1 and sets \textit{errno}. Any other value returned indicates success.
ConnectDetach_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:**

EINVAL     The connection specified by coid 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>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(*)
ConnectFlags(), ConnectFlags_r()
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:
  - `_NT0_COF_CLOEXEC` — close the connection if the process calls an `exec*()` function to start a new process.

Library:

- `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.
ConnectFlags(), ConnectFlags_r() © 2006, 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:

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 );

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()  © 2006, 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 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:

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 pid doesn’t have a connection >= coid.

ESRCH The process indicated by pid 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(), ConnectClientInfo(), MsgInfo(), MsgReceivev(), ND_NODE_CMP(), netmgr_ndtosstr(), netmgr_remote_node(), 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;
```
```c
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**

<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:**

significand()
Compute the cosine of an angle

Synopsis:

```c
#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

<table>
<thead>
<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()`, `errno`, `sin()`, `tan()`
**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:**

- `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 )
{
```
cosh(), coshf()

```c

    printf( "%f\n", cosh(.5) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
    1.127626
```

**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, 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:**

`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.

- **EINVAL** The call to `create()` 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.
**creat(), creat64()**

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;
    }
```

406  QNX Neutrino Functions and Macros  November 2, 2006
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()`, `execvp()`, `execve()`, `execvpe()`, `fcntl()`, `fileno()`,
`fstat()`, `isatty()`, `lseek()`, `open()`, `read()`, `sopen()`, `stat()`, `tell()`, `write()`,
`umask()`
crypt()

Encrypt a password

Synopsis:

```c
#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).
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>
<th></th>
</tr>
</thead>
<tbody>
<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 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 L.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

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>Read the Caveats</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
```

**This function: Is equivalent to calling:**

- `ctime()` `asctime( localtime( timer ) );`
- `ctime_r()` `asctime_r( localtime( timer ), buf );`

---

412  QNX Neutrino Functions and Macros  November 2, 2006
The `ctime()` function places the result string in a static buffer that’s reused each time you call `ctime()` or `asctime()`. Calling `gmtime()` or `localtime()` could also change the date in this static buffer.

The result string for `ctime_r()` is contained in the buffer pointed to by `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 `ctime()` or `ctime_r()` functions are called, the `tzset()` function is also called.

The calendar time is usually obtained by using the `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 `date` command to reflect Coordinated Universal Time (UTC), and then 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 string containing the formatted local time, or NULL if an error occurs.

**Classification:**

`ctime()` is ANSI, POSIX 1003.1; `ctime_r()` is POSIX 1003.1 TSF

**ctime()**

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

*continued…*
ctime(), ctime_r()

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

ctime_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>

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).

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:

#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:**

```c
libc
```

Use the `-l c` 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**

<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:**

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:

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

See also:

`DebugBreak()`, `DebugKDOutput()`
DebugKDOutput()

Print text with the kernel debugger

Synopsis:
#include <sys/neutrino.h>

void DebugKDOutput( 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…
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:

`DebugBreak()`, `DebugKDBreak()`
**delay()**

*Suspends 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()
devctl()
Control a device

Synopsis:

```c
#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:

```c
.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:
devctl()

- _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_FSY S, _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.
    */
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 CHG ON</td>
</tr>
<tr>
<td>.CTL_RTS.CHG</td>
<td>.CTL_RTS CHG OFF</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)
devctl()

- 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.

```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 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);
        }
    }
}
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):

```c
#include <termios.h>
#include <devctl.h>
#include <errno.h>
#include <sys/termios.h>

int tcdropline(int fd, int duration) {
    int error;
    duration = ((duration ? duration : 300) << 16) |
               _SERCTL_DTR_CTS |
               0;
    if(error = devctl(fd, DCMD_CHR_SERCTL, &duration, sizeof duration, 0) == -1) {
        if(error == ENOSTS) {
            errno = ENOTTY;
        }
        return -1;
    }
    return 0;
}
```
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:

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

<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:

asctime(), clock(), ctime(), gmtime(), localtime(), mktime(), strftime(), time(), tzset()
Synopsis:

```c
#include <dirent.h>

int dircntl(DIR *dir, int cmd, ...);
```

Arguments:

- `dir`: Provide control for this directory.
- `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.
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\n");
  } else {
    printf("Directory names are not filtered\n");
  }

  if(ret & D_FLAG_STAT) {
    printf("Servers asked for extra stat information\n");
  } else {
    printf("Servers not asked for extra stat information\n");
  }

  closedir(dp);
```
dircntl()

```c
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.

The string pointed to by `path` depends on where the application is run, so use `dirname()` in a portable way.

Note: This function isn’t thread-safe.

DAR

This function might modify the string pointed to by `path`, and can return a pointer to static storage.
dirname()

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>&quot;/usr/lib&quot;</td>
<td>&quot;/usr&quot;</td>
</tr>
<tr>
<td>&quot;/usr/&quot;</td>
<td>&quot;usr&quot;</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>&quot;..&quot;</td>
<td>&quot;..&quot;</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

<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…
dirname()

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:

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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**>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:

```c
dispatch_context_t *ctp, *new_c tp;

ctp = dispatch_context_alloc( ... );
while (1) {
    new_c tp = dispatch_block( ctp );
    if ( new_c tp ) {
        ctp = new_c tp
    } 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()

EINV AL 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(), and thread_pool_create().

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:

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
**dispatch_context_alloc()**

*Return a dispatch context*

**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

display_jump_to_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*.
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;
    }

    ctp = dispatch_context_alloc( dpp );

    return EXIT_SUCCESS;
}
```

For examples using the dispatch interface, see dispatch_create(), message_attach(), resmgr_attach(), and thread_pool_create().

Classification:

QNX Neutrino
**dispatch_context_alloc()**

<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:

`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*
**dispatch_context_free()**

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, "%s: Unable to allocate dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
```
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

<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_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.
Returns:

A handle to a dispatch structure, or NULL if an error occurs.

Errors:

ENOMEM    Insufficient memory to allocate context.

Examples:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <fcntl.h>
#include <sys/iocfunc.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],
    O_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

Cancellation point   No
Interrupt handler   No
Signal handler      No
Thread              Yes

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/iowrk.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.
Examples:

```c
#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, "%s: Unable to allocate dispatch handle.\n",argv[0] );
        return EXIT_FAILURE;
    }

    ...

    if ( (destroyed = dispatch_destroy ( dpp )) == -1 ) {  
        fprintf ( stderr, "Dispatch wasn't destroyed, \n bad dispatch handle %d.\n", dpp);
        return EXIT_FAILURE;
    }
/* else dispatch was destroyed */

    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:

\texttt{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 select_attach() was done and the pulse matches the one used by select_attach(), then the select_* subsystem is called and dispatches that event.

If a message is received, and no matching handler is found for that message type, MsgError() returns ENOSYS to the sender.

If a 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 sigwait_attach() function, not currently implemented). If a match is found, that function is called.

Returns:

0 Success.

-1 One of the following occurred:

- The message was a _PULSE_CODE_THREADDEATH pulse message (see 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 MSG_FLAG_CROSS_ENDIAN on this range, even though a handler was registered for it using message_attach(). The MSG_FLAG_CROSS_ENDIAN flag wasn’t given to 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 errno.
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

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Read the 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:

`dispatch_block()`, `dispatch_create()`, `dispatch_timeout()`

“Components of a resource manager” in the Writing a Resource Manager chapter of the Neutrino *Programmer’s Guide*
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 \n dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }

    ...

    if ( (timedout = dispatch_timeout ( dpp, &time_out ))
        == -1 ) {
        fprintf ( stderr, "Couldn’t set timeout \n" );
        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...


**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:**

`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()
**div()**

Calculate a quotient and remainder

**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 )
{
```
\begin{verbatim}
    div_t min_sec;
    min_sec = div( seconds, 60 );
    printf( "It took %d minutes and %d seconds\n",
            min_sec.quot, min_sec.rem );
}

int main( void )
{
    print_time( 130 );
    return EXIT_SUCCESS;
}
\end{verbatim}

produces the output:

\begin{verbatim}
It took 2 minutes and 10 seconds
\end{verbatim}

**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:**

`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.
dladdr()

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.

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>
</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 dlinfo 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:

dlclose(), dlerror(), dlopen(), dladdr()
dlclose()

Close a shared object

Synopsis:

```c
#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()`.

The `dlclose()` function is available only to dynamically linked processes.

Returns:

- 0 for success, or a nonzero value if an error occurs.

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()
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

<table>
<thead>
<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:

`dladdr()`, `dlclose()`, `dlopen()`, `dlsym()`
Synopsis:

```
#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 `RTLD_GLOBAL` flag. This set of symbols can change dynamically if the application subsequently calls `dlopen()` using `RTLD_GLOBAL`.

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:
  ```
  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:
**dlopen()**

RTLD_LAZY 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.

RTLD_NOW 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.

RTLD_LAZY isn’t currently supported.

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.

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.
Objects loaded with `dlopen()` that require relocations against global symbols can reference the symbols in any RTLD_GLOBAL 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 RTLD_GLOBAL 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 RTLD_GLOBAL 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 RTLD_GLOBAL 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:

```
qcc -Vgcc ntox86 -lE -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:**

```xml
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*
dlsym()

Get the address of a symbol in a shared object

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)();
}
```
dlsym()

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>

Caveats:

Function pointers are a pain; use typedefs to help preserve your sanity.

See also:

dladdr(), dlclose(), dlerror(), dlopen()
Synopsis:

```c
#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, `dn_comp()` updates the list of pointers for labels inserted into the message as the name is compressed. If dnptrs is NULL, names aren’t compressed. If lastdnptr is NULL, the list of labels isn’t updated.

**Returns:**

The size of the compressed domain 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_expand()`, `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 <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.
dn_expanded()  © 2006, QNX Software Systems GmbH & Co. KG.

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>
</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:

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>
</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>
drand48()

See also:

erand48(), jrand48(), lcong48(), lrand48(), mrand48(), nrand48(),.seed48(), srand48()
**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 \( r = x - n \cdot y \), 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, \( r = x - n \cdot y \), 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()*
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>
<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_deregister()`, `ds_flags()`
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.
ds_create()

Library:

libds

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

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`
Deregister an application with the data server

Synopsis:

```
#include <ds.h>

int ds_deregister( ds_t dsdes );
```

Arguments:

desdes 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.
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:

\textit{ds\_flags()}, \textit{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:

EBADF        Invalid file descriptor dsdes.
ESRCH        The variable doesn’t exist in the data server.

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_clear(), ds_create(), ds_deregister(), 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**

<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_deregister()*
ds_set()

Set a data server variable

Synopsis:

```
#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:

- `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.
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

<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_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;
```
**dup()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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(),
exclp(), execlpe(), execv(), execve(), execvp(), execvepe(), fcntl(),
fileno(), fstat(), isatty(), lseek(), open(), read(), sopen(), stat(), tell(),
umask(), write()
**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:
dup2()

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;
}

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:

chsize(), close(), creat(), dup(), eof(), errno, execl(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(), fcntl(), fileno(), fstat(), isatty(), lseek(), open(), read(), sopen(), stat(), tell(), umask(), write()
**Synopsis:**

```c
#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.
  - 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 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
Cancellation point   No
Interrupt handler    No

continued...
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()
Encrypt a string

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.

Note: 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

---

520  QNX Neutrino Functions and Macros  November 2, 2006
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()
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()
Synopsis:

```c
#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

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:

`gethostbyaddr()`, `gethostbyname()`, `gethostent()`, `hostent`, `sethostent()`
endhostent()
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

<table>
<thead>
<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_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()`,
`nthos()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`,
`UNALIGNED_RET64()`, `UNALIGNED_PUT16()`,
`UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
**Synopsis:**

```c
#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:

```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_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()
ENDIAN_BE64()

Return a big-endian 64-bit value in native format

Synopsis:

```c
#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:

```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_BE64( val ) );

    return EXIT_SUCCESS;
}
```
On a little-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_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()

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()`
Synopsis:

```
#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:

```
#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;
}
```
**ENDIAN_LE32()**

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_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:

```c
#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:

```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_LE64( val ) );

    return EXIT_SUCCESS;
}
```
ENDIAN_LE64()

On a big-endian system, this prints:

\[0x1234\text{deadbeef5678} = 0x7856\text{efbeadde3412}\]

**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:**

`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:
#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:

#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>
</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\_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()**
Synopsis:
#include <gulliver.h>

uint32_t ENDIAN_RET32( uint32_t num );

Arguments:

num  The number you want to convert.

Library:

libc
Use the -lc 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:

#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_RET32( val ) );
    return EXIT_SUCCESS;
}
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 RET32()` is implemented as a macro.

**See also:**

`ENDIAN_BE16()`, `ENDIAN_BE32()`, `ENDIAN_BE64()`,
`ENDIAN_LE16()`, `ENDIAN_LE32()`, `ENDIAN_LE64()`,
`ENDIAN_RET16()`, `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_RET64()

Return an endian-swapped 64-bit value

Synopsis:

```
#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:

```
#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:

\[0x1234\text{deadbeef}5678 = 0x7856\text{efbeadde}3412\]

**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\_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()`,
`ntohs()`, `UNALIGNED\_RET16()`, `UNALIGNED\_RET32()`,
`UNALIGNED\_RET64()`, `UNALIGNED\_PUT16()`,
`UNALIGNED\_PUT32()`, `UNALIGNED\_PUT64()`
Synopsis:

```c
#include <gulliver.h>

void ENDIAN_SWAP16( uint16_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_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:

```
val = 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_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()`, `ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`, `UNALIGNED_RET64()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
**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_SWAP64()`, `htonl()`, `htons()`, `ntohl()`, `ntohs()`, `UNALIGNED_RET16()`, `UNALIGNED_RET32()`, `UNALIGNED_RET64()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`, `UNALIGNED_PUT64()`
Synopsis:

#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 endi-an-swaps the value pointed to by num in place.

Examples:

Swap the endianness of a value:

#include <stdio.h>
#include <stdlib.h>
#include <gulliver.h>
#include <inttypes.h>

int main( void )
{
    uint64_t val = 0x1234deadbeef5678LL;
    ENDIAN_SWAP64( &val );

    printf( "val = 0x%016x\n", val );

    return EXIT_SUCCESS;
}

This prints:

val = 0x7856efbeadde3412
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_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()`.
endnetent()

Close the network name database file

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>
</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:

`getnetbyaddr()`, `getnetbyname()`, `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:

```c
libsocket
```

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

Description:

The `endprotoent()` routine closes the protocol name database file.

Files:

```c
/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>

See also:

`getprotobyname()`, `getprotobynumber()`, `getprotoent()`, `protoent()`, `setprotoent()`
/etc/protocols in the Utilities Reference
endpwent()

Close the password database file

Synopsis:

```c
#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
**Synopsis:**

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

void endspent( void );
```

**Library:**

```c
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**

<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:

getspent(), setspent()
Synopsis:

```c
#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>
See also:

getutent(), getutid(), getutline(), pututline(), setutent(), utmp, utmpname()

login in the Utilities Reference
**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()`
**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

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
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
erand48()

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:

drand48(), jrand48(), lcong48(), lrand48(), mrand48(), nrand48(), seed48(), srand48()
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:

```c
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: \( \text{erf}(-x) = -\text{erf}(x) \)

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`. 
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:**

`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.
err(), errx()

<table>
<thead>
<tr>
<th>Function</th>
<th>errno string?</th>
<th>Varargs?</th>
<th>Exits?</th>
</tr>
</thead>
<tbody>
<tr>
<td>err()</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>errx()</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>verr()</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>verrx()</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>vwarn()</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>vwarnx()</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>warn()</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>warnx()</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

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.
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()`
**Synopsis:**

```
#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_errlist`: An array of error messages corresponding to `errno`.
- `sys_nerr`: The number of entries in the `sys_errlist` 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>EADDRNOTA</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 protocol family</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Resource is temporarily unavailable; try 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>
</tbody>
</table>

*continued…*
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>EBUSY</td>
<td>Device or resource is busy</td>
</tr>
<tr>
<td>ECANCELED</td>
<td>Operation canceled (1003.1b-1993)</td>
</tr>
<tr>
<td>ECHILD</td>
<td>No child processes</td>
</tr>
<tr>
<td>ECHRNG</td>
<td>Channel number is out of range</td>
</tr>
<tr>
<td>ECOMM</td>
<td>Communication error occurred on send</td>
</tr>
<tr>
<td>ECONNABORTED</td>
<td>Software caused connection to abort</td>
</tr>
<tr>
<td>ECONNREFUSED</td>
<td>Connection refused</td>
</tr>
<tr>
<td>ECONNRESET</td>
<td>Connection reset by peer</td>
</tr>
<tr>
<td>ECTRLTERM</td>
<td>Remap to the controlling terminal</td>
</tr>
<tr>
<td>EDEADLK</td>
<td>Resource deadlock avoided</td>
</tr>
<tr>
<td>EDEADLOCK</td>
<td>File locking deadlock</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>Destination address is required</td>
</tr>
<tr>
<td>EDOM</td>
<td>Math argument is out of domain for the function</td>
</tr>
<tr>
<td>EDQUOT</td>
<td>Disk quota exceeded</td>
</tr>
<tr>
<td>EEXIST</td>
<td>File exists</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Bad address</td>
</tr>
</tbody>
</table>

continued…
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFBIG</td>
<td>File is too large</td>
</tr>
<tr>
<td>EHOSTDOWN</td>
<td>Host is down</td>
</tr>
<tr>
<td>EHOSTUNREACH</td>
<td>Unable to communicate with remote node</td>
</tr>
<tr>
<td>EIDRM</td>
<td>Identifier removed</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>Illegal byte sequence</td>
</tr>
<tr>
<td>EINPROGRESS</td>
<td>Operation now in progress</td>
</tr>
<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>
</tbody>
</table>

*continued*
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMLINK</td>
<td>Too many links</td>
</tr>
<tr>
<td>MORE</td>
<td>More to do, send message again</td>
</tr>
<tr>
<td>EMSGSIZE</td>
<td>Inappropriate message buffer length</td>
</tr>
<tr>
<td>MULTIHOP</td>
<td>Multihop attempted</td>
</tr>
<tr>
<td>ENAME_TOO_LONG</td>
<td>Filename is too long</td>
</tr>
<tr>
<td>NETDOWN</td>
<td>Network is down</td>
</tr>
<tr>
<td>NETRESET</td>
<td>Network dropped connection on reset</td>
</tr>
<tr>
<td>NETUNREACH</td>
<td>Network is unreachable</td>
</tr>
<tr>
<td>ENFILE</td>
<td>Too many open files in the system</td>
</tr>
<tr>
<td>NOANO</td>
<td>No anode</td>
</tr>
<tr>
<td>NOBUFS</td>
<td>No buffer space available</td>
</tr>
<tr>
<td>NOCSI</td>
<td>No CSI structure available</td>
</tr>
<tr>
<td>NODATA</td>
<td>No data (for no-delay I/O)</td>
</tr>
<tr>
<td>NODEV</td>
<td>No such device</td>
</tr>
<tr>
<td>NOENT</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>NOEXEC</td>
<td>Exec format error</td>
</tr>
<tr>
<td>NOLCK</td>
<td>No locks available</td>
</tr>
<tr>
<td>NOLIC</td>
<td>No license available</td>
</tr>
<tr>
<td>NOLINK</td>
<td>The link has been severed</td>
</tr>
<tr>
<td>NOMEM</td>
<td>Not enough memory</td>
</tr>
<tr>
<td>NOMSG</td>
<td>No message of desired type</td>
</tr>
<tr>
<td>NONDP</td>
<td>Need an NDP (8087...) to run</td>
</tr>
<tr>
<td>NONET</td>
<td>Machine isn’t on the network</td>
</tr>
</tbody>
</table>

*continued...*
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

*continued...*
## errno

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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>EW OULDBLOCK</td>
<td>Operation would block</td>
</tr>
<tr>
<td>EXDEV</td>
<td>Cross-device link</td>
</tr>
<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 )
{
    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:

h_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:

\texttt{brk()}, \texttt{\_btext}, \texttt{\_edata}, \texttt{\_end}, \texttt{sbrk()}
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:
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 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.

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 statime field of the file is marked for update. If the exec* function failed but was able to locate the process image file, whether the statime 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 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>
<tr>
<td>execlpe()</td>
<td>NULL-terminated argument list, search for the new process in PATH, specify the new process environment</td>
<td>No</td>
</tr>
</tbody>
</table>

continued...
**execv()**

© 2006, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>POSIX?</th>
</tr>
</thead>
<tbody>
<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 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 environment</td>
<td>No</td>
</tr>
</tbody>
</table>

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 myprog as if a user had typed:

myprog ARG1 ARG2

at the shell:

#include <stddef.h>
#include <process.h>

excl( "myprog", "myprog", "ARG1", "ARG2", NULL );

In this example, myprog will be found if it exists in the current working directory.

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:

abort(), atexit(), errno, execle(), execlp(), execle(), execvp(), execve(),
execvp(), execvpe(), _exit(), exit(), getenv(), main(), putenv(), spawn(),

spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), 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 `fentl()`’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><code>excl()</code></td>
<td>NULL-terminated argument list</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>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>
<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 <stddef.h>
#include <process.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.
execle()  

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:

abort(), atexit(), errno, execl(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(), _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawne(), spawnvp(), spawnvpe(), system()
**Synopsis:**

```
#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 `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 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><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>
<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.

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.

Classification:
POSIX 1003.1

Safety
Cancellation point No
Interrupt handler No
Signal handler No
Thread Yes

See also:
abort(), atexit(), errno, execl(), execlp(), execle(), execvp(), execve(), execvp(), execvpe() _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system()
execlepe()

Synopsis:

```
#include <process.h>

int execlepe( 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.
Description:

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>execcl()</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>
</tbody>
</table>

*continued…*
Function | Description | POSIX?
---|---|---
`execve()` | NULL-terminated array of arguments, specify the new process’s environment | Yes
`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 `execlpe()` 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.
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:

abort(), atexit(), errno, execl(), execle(), execlp(), execv(), execve(), execvp(), execvpe(), _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnp(), spawnve(), spawnvpe(), system()
Synopsis:
#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:

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())
• `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…
execv()

<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 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 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.
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>
<th></th>
</tr>
</thead>
<tbody>
<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()`, `execve()`, `execvp()`, `execvep()`, `_exit()`, `exit()`, `getenv()`, `main()`, `putenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`, `system()`
execve()

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:
int main (int argc, char *argv[]);

where \texttt{argc} is the argument count and \texttt{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 \texttt{argv} and \texttt{environ} arrays are each terminated by a null pointer. The null pointer terminating the \texttt{argv} array isn’t counted in \texttt{argc}.

Multithreaded applications shouldn’t use the \texttt{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 \texttt{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 \texttt{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 \texttt{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*` 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>
</tbody>
</table>

*continued...*
execve()

Function | Description | POSIX?
--- | --- | ---
execv() | NULL-terminated array of arguments | Yes
execve() | NULL-terminated array of arguments, specify the new process’s environment | Yes
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 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.
execve()

ENOMEM There’s insufficient memory available to create the new process.

ENOTDIR A component of path isn’t a directory.

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(), errno, execl(), execle(), execlpe(), execv(), execvp(), execve(), _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system()
Synopsis:

```c
#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:

```
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
execvp()

- 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>execl()</td>
<td>NULL-terminated argument list, specify the new process’s environment</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
execvp()

<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 <strong>PATH</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><code>execlpe()</code></td>
<td>NULL-terminated argument list, search for the new process in <strong>PATH</strong>, 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 <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 `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.
**execvp()**

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.

**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:**

`abort()`, `atexit()`, `errno`, `execl()`, `execlp()`, `execlpe()`, `execv()`, `execve()`, `execvpe()`, `_exit()`, `exit()`, `getenv()`, `main()`, `putenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`, `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>
</tbody>
</table>

*continued.*
Function | Description | POSIX?
--- | --- | ---
execve() | NULL-terminated array of arguments, specify the new process’s environment | Yes
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 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 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(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), _exit(), exit(), getenv(), main(), putenv(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), 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.

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.
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.

Sends a SIGCHLD signal to the parent process.

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.

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.

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(), execl(), execl(), execlp(), execle(), execvp(),
execve(), execvpe(), exit(), getenv(), main(), putenv(),
sigaction(), signal(), spawn(), spawnl(), spawnle(), spawnlp(),
spawnlpe(), spawnp(), 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.

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

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
See also:

abort(), atexit(), _exit(), main()
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:**

- `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

<table>
<thead>
<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 value of \( \text{expm1}(x) \) may be more accurate than \( \text{exp}(x) - 1.0 \) for small values of \( x \).

**See also:**

\( \text{errno}, \text{expm1}, \text{log()} \)
**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:

\[
\text{expm1}(n \times \log1p(x))/x
\]

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 \( \text{errno} \) to 0, call the function, and then check \( \text{errno} \) again. These functions don't change \( \text{errno} \) 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

continued…
expm1(), expm1f()

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>

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>
 Computes 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( "%f %f\n", fabs(.5), fabs(-.5) );
    return EXIT_SUCCESS;
}
```
produces the output:

0.500000 0.500000

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:

abs(), cabs(), labs()
Synopsis:

```
#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 `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 `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>
<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:**

`cfgopen()`, `confstr()`

`mib.txt`, `snmpd.conf` in the *Utilities Reference*
fchmod()  Change the permissions for a file

Synopsis:

#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] );
        }
```
```c
ecode++;
}
else if( fchmod( fd, S_IRUSR | S_IWUSR ) == -1 ) {
    perror( argv[i] );
    ecode++;
}

    close( fd );
}
return ecode;

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:

`chmod()`, `chown()`, `errno`, `fchown()`, `fstat()`, `open()`, `stat()`
Synopsis:

```c
#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] );
    }
  }
  return ecode;
}
```


```
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>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()`, `chown()`, `errno`, `fchmod()`, `fstat()`, `lchown()`, `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()

    return EXIT_FAILURE;
    }

Classification:

    ANSI, POSIX 1003.1

Safety

    Cancellation point   Yes
    Interrupt handler   No
    Signal handler      No
    Thread              Yes

See also:

    errno, fcloseall(), fdopen(), fopen(), freopen()
fcloseall()

Close all open stream files

Synopsis:

```
#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:

```
#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

<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:

fclose(), fdopen(), fopen(), freopen()
**fcntl()**

Provide control over an open file

**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:

- **F_ALLOCSP** 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.
- **F_DUPFD** 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 `int`. The new file descriptor refers to the same file as `fildes`, and shares any locks.

**F_FREESP**  Truncate the file to the size (in bytes) specified by the extra argument, `arg`.

**F_GETFD**  Get the file descriptor flags associated with the file descriptor `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 `fildes`. The flags and modes are defined in `<fcntl.h>`.

The file status flags (see `open()` for more detailed information) are:

- **O_APPEND** — Set append mode.
- **O_NONBLOCK** — No delay.

The file access modes are:

- **O_RDONLY** — Open for reading only.
- **O_RDWR** — Open for reading and writing.
- **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 `struct flock` (defined in `<fcntl.h>`). For more information, see the `flock` structure section below. The information returned overwrites the information passed to `fcntl()` in the structure pointed to by `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 `F_UNLCK`. If a lock is
found, the \texttt{lpid} member of the structure pointed to by \texttt{arg} is set to the process ID of the process holding the blocking lock and \texttt{l whence} is set to \texttt{SEEK_SET}.

F\textunderscore SETFD

Set the file descriptor flags associated with \texttt{fildes} to the third argument, \texttt{arg}, taken as type \texttt{int}. See the above discussion for more details.

The only defined file descriptor flag is:

\texttt{FD\_CLOEXEC} When this flag is clear, the file remains open across \texttt{spawn}\textsuperscript{*}() or \texttt{exec}\textsuperscript{*}() calls; else the file is closed.

F\textunderscore SETFL

Set the file status flags, as shown above, for the open file description associated with \texttt{fildes} from the corresponding bits in the third argument, \texttt{arg}, taken as type \texttt{int}. You can’t use this function to change the file access mode. All bits set in \texttt{arg}, other than the file status bits, are ignored.

F\textunderscore SETLK

Set or clear a file segment lock, according to the lock description pointed to by the third argument, \texttt{arg}, taken as a pointer to type \texttt{struct flock}, as defined in the header file \texttt{<fcntl.h>}, and documented below. This command is used to create the following locks (defined in \texttt{<fcntl.h>}):

\texttt{F\_RDLOCK} Shared or read locks.
\texttt{F\_UNLOCK} Remove either type of lock.
\texttt{F\_WRLOCK} Exclusive or write locks.

If a lock can’t be set, \texttt{fcntl()} returns immediately.

F\textunderscore SETLK\textunderscore W

This command is the same as F\textunderscore 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 \texttt{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 \( l_{\text{len}} \) is set to zero. If \( l_{\text{whence}} \) and \( l_{\text{start}} \) point to the beginning of the file, and \( l_{\text{len}} \) 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 \texttt{fork()} function. However, locks are inherited across \texttt{exec*()} or \texttt{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, \texttt{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 \textit{cmd} is \texttt{F_SETLK}, the type of lock (\texttt{l_type}) is a shared lock (\texttt{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 \texttt{fildes} argument isn’t a valid file descriptor.
The argument \textit{cmd} is \texttt{F_SETLK} or \texttt{F_SETLKW}, the type of lock (\texttt{l_type}) is a shared lock (\texttt{F_RDLCK}), and \texttt{fildes} isn’t a valid file descriptor open for reading.

The argument \textit{cmd} is \texttt{F_SETLK} or \texttt{F_SETLKW}, the type of lock (\texttt{l_type}) is an exclusive lock (\texttt{F_WRLCK}), and \texttt{fildes} isn’t a valid file descriptor open for writing.

EDEADLK The argument \textit{cmd} is \texttt{F_SETLKW}, and a deadlock condition was detected.

EINTR The argument \textit{cmd} is \texttt{F_SETLKW}, and the function was interrupted by a signal.

EINVAL The argument \textit{cmd} is \texttt{F_DUPFD}, and the third argument is negative, or greater than the configured number of maximum open file descriptors per process.
The argument \textit{cmd} is \texttt{F_GETLK}, \texttt{F_SETLK} or \texttt{F_SETLKW}, and the data \textit{arg} isn’t valid, or \texttt{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:**

```c
/*
 * 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Read the Caveats</td>
</tr>
<tr>
<td>Interrupt handler</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 `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()`, `execl()`, `execle()`, `execle()`, `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.
The `fdatasync()` function isn’t supported for the filesystem specified by `filedes`.

**Classification:**

POSIX 1003.1 SIO

<table>
<thead>
<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:**

`aio_fsync()`, `close()`, `fcntl()`, `fsync()`, `open()`, `read()`, `sync()`, `write()`
Synopsis:

```c
#include <stdio.h>

FILE* fdopen( int filedes,
             const char* mode );
```

Arguments:

- `filedes` The file descriptor that you want to associate with a stream.
- `mode` The mode specified when `filedes` 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 `filedes`, which represents an opened file or device.

The `filedes` argument is a file descriptor that was returned by one of `accept()`, `creat()`, `dup()`, `dup2()`, `fcntl()`, `open()`, `pipe()`, or `sopen()`.

The `fdopen()` function preserves the offset maximum previously set for the open file description corresponding to `filedes`.

Returns:

A file stream for success, or `NULL` if an error occurs (`errno` is set).
Errors:

EBADF The `filedes` argument isn’t a valid file descriptor.

EINVAL The `mode` argument isn’t a valid mode.

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

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>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()`
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 )
```

674  C Library — F to H  November 2, 2006
{ 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(),
fopen(), freopen(), getc(), getc_unlocked(), getchar(),
getc_unlocked(), gets(), getw(), getwc(), getwchar(), perror(),
read()
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;
}
```
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:

clearerr(), feof(), fgetc(), fgetchar(), fgets(), fgetwc(), fgetws(), getc(),
getc_unlocked(), getchar(), getchar_unlocked(), gets(), getw(),
getwc(), getwchar(), perror(), strerror()
Flush the buffers for a stream

Synopsis:

```c
#include <stdio.h>

int fflush( FILE* fp );
```

Arguments:

- `fp` 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()
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 ) {
    fputc( c, stdout );
}
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:

errno, feof(), ferror(), fgetchar(), fgets(), fopen(), fputc(), getc(), gets(), ungetc()
Synopsis:

```c
#include <stdio.h>

int fgetchar( void );
```

Library:

```none
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, stdout );
        }
    }
    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>

See also:

errno, feof(), ferror(), fgetc(), fputchar(), getc(), getchar()
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:

`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];
    // Example code...

    // Call fgetpos()...
    // ...and fsetpos()...

    // Output or use...
}
```
fgetpos()

```c
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.
Don’t assume that there’s a newline character in every string that you read with \texttt{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.

\begin{description}
\item[Returns:] The same pointer as \texttt{buf}, or NULL if the stream is at the end-of-file or an error occurs (\texttt{errno} is set).
\item[Use] Use \texttt{feof()} or \texttt{ferror()} to distinguish an end-of-file condition from an error.
\end{description}

\begin{quote}
\textbf{Examples:}

\begin{verbatim}
#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;
}
\end{verbatim}
\end{quote}
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(), fopen(), fputs(), getc(), gets(), fgets()
**fgetspent()**

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()` function 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()`
- `fopen()`
- `fseek()`
- `rewind()`

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
", 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:
#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()**

Either a physical I/O error has occurred, or the process is in the background and is being ignored or blocked.

Cannot read at or beyond the offset maximum for this stream.

**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`, `feof()`, `ferror()`, `fputwc()`

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter
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:

- `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 (errno is set).

buf Success.

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.

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(), fputws()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
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);
```
finite(), finitef()

```c
void
{ 
    double a = 2.0; 
    double b = -0.5; 
    double c = NaN; 
    double d = Inf; 
    
    d = 1.0/0.0; 
    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**

<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:**

`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.
flink()

ELOOP          Too many levels of symbolic links or prefixes.
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>

© 2006, QNX Software Systems GmbH & Co. KG.
See also:

$link()$
Apply or remove an advisory lock on an open file

Synopsis:

```c
#include <fcntl.h>

int flock( int filedes,
           int operation );
```

Arguments:

- `filedes`  
The file descriptor of an open file.
- `operation`  
What you want to do to the file; see below.

Library:

`libc`

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>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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(), lockf(), open()
$flockfile()$

Acquire ownership of a file

Synopsis:

```c
#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.
flockfile()

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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

getc_unlocked(), getchar_unlocked(), putc_unlocked(), putchar_unlocked()
floor(), floorf()
Round down a value to the next integer

Synopsis:

```
#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 \( \leq x \) (rounding towards the “floor”).

Returns:

The largest integer \( \leq x \).

---

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:

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

int main( void )
{
    printf( "%f\n", floor( -3.14 ) );
}
```
```c
printf( "\n", floor( -3. ) );
printf( "\n", floor( 0. ) );
printf( "\n", floor( 3.14 ) );
printf( "\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()`
flushall()  
Flush all input/output buffers

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()

© 2006, QNX Software Systems GmbH & Co. KG.

Caveats:

The QNX 4 version of this function returns the number of streams flushed.

See also:

errno, fopen(), flush()
**fmod(), fmodf()**

*Compute a residue, using floating-point modular arithmetic*

**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:**

```c
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 `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", fmod( 4.5, 2.0 ) );
    printf( "%.5f\n", fmod( -4.5, 2.0 ) );
    printf( "%.5f\n", fmod( 4.5, -2.0 ) );
    printf( "%.5f\n", fmod( -4.5, -2.0 ) );

    return EXIT_SUCCESS;
}
```

produces the output:

```
0.50000
-0.50000
0.50000
-0.50000
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<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:

\texttt{ceil()}, \texttt{div()}, \texttt{fabs()}, \texttt{floor()}
fnmatch() © 2006, QNX Software Systems GmbH & Co. KG.

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:

  **FNMPATHNAME**
  - 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.

  **FNMPERIOD**
  - If this is set, a leading period in `str` matches a period in `pat`, where the definition of “leading” depends on FNMPATHNAME:
    - If FNMPATHNAME is set, a period is leading if it’s the first character in `str`, or if it immediately follows a slash.
    - If FNMPATHNAME isn’t set, a period is leading only if it’s the first character in `str`.

  **FNQUOTE**
  - If this isn’t set, a backslash (\) in `pat` followed by another character matches that second character. If FNQUOTE is set, a backslash is treated as an ordinary character.
fnmatch()

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 \texttt{a[bc]} matches the strings \texttt{ab} and \texttt{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 \texttt{a*d} matches the strings \texttt{ad}, \texttt{abd}, and \texttt{abcd}, but not the string \texttt{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 \texttt{a*d*} matches the strings \texttt{ad}, \texttt{abcd}, \texttt{abcdef}, \texttt{aaaad} and \texttt{adddd}; the pattern \texttt{*a*d} matches the strings \texttt{ad}, \texttt{abcd}, \texttt{efabcd}, \texttt{aaaad} and \texttt{adddd}.

**Returns:**

- \texttt{0} The \texttt{str} argument matches the pattern specified by \texttt{pat}.
- Nonzero The \texttt{str} argument doesn’t match the pattern specified by \texttt{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()

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:

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:

```
lbc
```

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.
**fopen(), fopen64()**

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
**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, 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.
fork()

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.
- **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 ) {
        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 );
}
fork()

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(), execl(), execlp(), execle(), execv(), execve(), execvp(), execve(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), wait()
forkpty()

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

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:

fork(), login_jty(), openpty(), termios
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 \texttt{fp\_exception\_mask()} function gets or sets the current exception mask, depending on the value of the \textit{set} argument.

Returns:

<table>
<thead>
<tr>
<th>set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>The current exception mask.</td>
</tr>
<tr>
<td>≥ 0</td>
<td>The previous mask.</td>
</tr>
</tbody>
</table>

This function doesn’t return a special value to indicate that an error occurred. If you want to check for errors, set \textit{errno} to 0, call the function, and then check \textit{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("Invalid ");
printf("\n");
return(0);
}

Classification:

QNX Neutrino

Safety

<table>
<thead>
<tr>
<th></th>
<th>No</th>
<th>Yes</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:

*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>`.

© 2006, QNX Software Systems GmbH & Co. KG.
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()`
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 if (ret == _FP_PREC_DOUBLE_EXTENDED)
        printf("128 Bit \n");
    else
        printf("Error \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_mask(), fp_exception_value(), fp_rounding()
fp_rounding()
Set or get the current 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 \textit{-l m} option to \texttt{qcc} to link against this library.

Description:
The \textit{fp_rounding()} function sets or gets the current rounding mode,
depending on the value of \textit{newrounding}.

Returns:
If \textit{newrounding} is less than 0, the current rounding mode; otherwise,
the previous mode.

\footnote{This function doesn't return a special value to indicate that an error
occurred. If you want to check for errors, set \textit{errno} to 0, call the
function, and then check \textit{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 \n");
    else if (ret == _FP_ROUND_POSITIVE)
        printf("Positive \n");
    else if (ret == _FP_ROUND_NEGATIVE)
        printf("Negative \n");
    else if (ret == _FP_ROUND_ZERO)
        printf("To Zero \n");
    else
        printf("Error \n");

    return EXIT_SUCCESS;
}
```

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:

\( \text{fp\_exception\_mask()}, \text{fp\_exception\_value()}, \text{fp\_precision()} \)
**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:**

`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).

_PCPIPE_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>
</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:

`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\n",
    weekday, month, 10, 1999 );

return EXIT_SUCCESS;
}

Produces:

Saturday, April 10, 1999

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, fwprintf(), printf(), snprintf(), sprintf(), swprintf(), 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 ) {
            // Do something with c
        }
    }
}
```
```c
fputc( c, stdout );
}
fclose( fp );
return EXIT_SUCCESS;
}
return EXIT_FAILURE;
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

**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()`, `fgets()`, `putc()`, `putchar()`, `puts()`
fputchar()  © 2006, QNX Software Systems GmbH & Co. KG.
Write a character to stdout

Synopsis:

```
#include <stdio.h>

int fputchar( int c );
```

Arguments:

- **c**  The character 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 `fputchar()` function writes the character specified by `c`, cast as `(int)(unsigned char)`, to `stdout`. It’s equivalent to `putchar()` and to:

```
fputc( c, stdout );
```

Returns:

The character written, cast as `(int)(unsigned char)`, or EOF if an error occurred (`errno` is set).

Examples:

```
#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 ) {
        fputchar( c );
        c = fgetc( fp );
    }
    fclose( fp );

    return EXIT_SUCCESS;
}

return EXIT_FAILURE;

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>

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()
fputs() 
Write a string to an output stream

Synopsis:
#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.

☞ The terminating NUL character isn’t written.

Returns:
A nonnegative value for success, or EOF if an error occurs (errno is set).

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

int main( void )
{
    FILE *fp_in, *fp_out;
    char buffer[80];
}
fputs()

```c
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**

<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, fgets(), fopen(), fprintf(), fputc(), putc(), 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()

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFBIG</td>
<td>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.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal terminated the write operation; no data was transferred.</td>
</tr>
</tbody>
</table>
| 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>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, fgetswc(), fputws()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
**fputws()**

Write a wide-character string to an output stream

**Synopsis:**

```c
#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`.

☞ 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.

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, fgets(), fputwc()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
fread()

Read elements of a given size from a stream

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

<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`, `fopen()`, `feof()`, `ferror()`
Synopsis:
#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:

#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

**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:**

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()`
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

<table>
<thead>
<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:

`addrinfo`, `gai_strerror()`, `getaddrinfo()`
freeifaddrs()
Free a network interface address

Synopsis:

```
#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>
</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:

`getifaddrs()`, `ifaddrs`, `ioctl()`, `malloc()`, `socket()`, `sysctl()`
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.
freopen(), freopen64()

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()).

EINTR       The freopen() 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 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:

```c
#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.
```
```c
/*
 fp = freopen( "file", "r", stdin );

 if( fp != NULL ) { /* Now we can read from "file" using the
 * stdin functions like fgetchar()...
 */
 while( ( c = fgetchar() ) != EOF ) {
   fputc( c );
 }

close( fp );
 return EXIT_SUCCESS;
}

return EXIT_FAILURE;
*/
```

**Classification:**

`freopen()` is ANSI, POSIX 1003.1; `freopen64()` is Large-file support

**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`, `fclose`, `fcloseall`, `fdopen`, `fopen`, `fopen64()`
frexp(), frexpf()

Break a floating-point number into a normalized fraction and an integral power of 2

Synopsis:

```
#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:

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

int main( void )
```
frexp(), frexpf() © 2006, QNX Software Systems GmbH & Co. KG.

```c
{
    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**

<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:**

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 )
{
    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(), sswscanf(), vfscanf(), vfswscanf(),
vscanf(), vsscanf(), vswscanf(), vvscanf(), wscanf()
Synopsis:

```c
#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. */
```
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>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, fgetpos(), fopen(), fsetpos(), ftell()
fsetpos()
Set the current position of a file

Synopsis:

```
#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()**

<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, 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.
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.
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:

\textit{fstat()} is POSIX 1003.1; \textit{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:

\textit{creat()}, \textit{dup()}, \textit{dup2()}, \textit{errno}, \textit{fcntl()}, \textit{lstat()}, \textit{open()}, \textit{pipe()}, \textit{sopen()}, \textit{stat()}
fstatvfs(), fstatvfs64()

Get filesystem information, given a file descriptor

Synopsis:

```
#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 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.
**fstatvfs(), fstatvfs64()**

**Returns:**

0  Success.

-1  An error occurred (errno is set).

**Errors:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The fildes argument isn’t an open file descriptor.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The buf argument points to an illegal address.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal was caught during execution.</td>
</tr>
<tr>
<td>EIO</td>
<td>An I/O error occurred while reading the filesystem.</td>
</tr>
<tr>
<td>EOVERFLOW</td>
<td>One of the values to be returned can’t be represented correctly in the structure pointed to by buf.</td>
</tr>
</tbody>
</table>

**Classification:**

fstatvfs() is POSIX 1003.1 XSI; fstatvfs64() is Large-file support

**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>

**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()
**fsync()**

**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()

**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()`
ftell(), ftello()

Return the current position of a stream

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 )
{
```

792  C Library — F to H  November 2, 2006
```c
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. */
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:**

`ftell()` is ANSI, POSIX 1003.1; `ftello()` is POSIX 1003.1

**Safety**

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

`continued...`
ftell(), ftello()

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

See also:

errno, fgetpos(), fopen(), fsetpos(), fseek()
ftime()
Get the current time

Synopsis:

```c
#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()
**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_ctime` and `st_mtime` fields of the file for update. If the `ftruncate()` function is unsuccessful, the file is unaffected.
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

<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{mmap()}, \textit{open()}, \textit{shm\_open()}, \textit{truncate()}
**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**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

`flockfile()`, `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()`,
`putchar_unlocked()`
ftw(), ftw64()

Walk a file tree

Synopsis:

```
#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:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTW_F</td>
<td>The object is a file.</td>
</tr>
<tr>
<td>FTW_D</td>
<td>The object is a directory.</td>
</tr>
<tr>
<td>FTW_DNR</td>
<td>The object is a directory that can’t be read. Descendents of the directory aren’t processed.</td>
</tr>
<tr>
<td>FTW_NS</td>
<td>The <code>stat()</code> failed on the object because the permissions weren’t appropriate, or the object is a symbolic link that points to a nonexistent file. The <code>stat</code> buffer passed to <code>fn()</code> is undefined.</td>
</tr>
</tbody>
</table>

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

<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:

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:

```c
#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

<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>
funlockfile()

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 \texttt{futime()} function in this way.

\textbf{Returns:}

\begin{itemize}
  \item [0] Success.
  \item [-1] An error occurred (\texttt{errno}) is set.
\end{itemize}

\textbf{Errors:}

- \textbf{EACCES} Search permission is denied for a component of \texttt{path},
or the \texttt{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.

- \textbf{ENAMETOOLONG} The argument \texttt{path} exceeds \texttt{PATH_MAX} in length, or a
pathname component is longer than \texttt{NAME_MAX}.

- \textbf{ENOENT} The specified \texttt{path} doesn’t exist, or \texttt{path} is an empty
string.

- \textbf{ENOTDIR} A component of \texttt{path} isn’t a directory.

- \textbf{EPERM} The \texttt{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.

- \textbf{EROFS} The named file resides on a read-only filesystem.

\textbf{Classification:}

Unix
futime()

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, utime()
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 \( fp \) argument isn’t valid.

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>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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>SafetyFeature</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()
**Synopsis:**

```c
#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
fwrite() 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, ferror(), fopen()
fwscanf() (C Library — F to H)

Scan wide-character input from a stream

Synopsis:

#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()

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, fscanf(), scanf(), sscanf(), swscanf(), vfscanf(), vfwscanf(), vscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()`
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.
- `EAIAGAIN` 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()*


Log gamma function

Synopsis:

```c
#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 Γ(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|Γ(x)|, where Γ(x) is defined as follows:

For `x > 0`:

\[ \int_0^\infty e^{-t} t^{x-1} dt \]

For `x < 1`:

\[ \frac{n}{\Gamma(1-x) \cdot \sin(nx)} \]
The results converge when \( x \) is between 0 and 1. The Gamma function has the property:

\[
\Gamma(N) = \Gamma(N-1) \times N
\]

The `gamma*` functions compute the log because the Gamma function grows very quickly.

The `gamma()` and `gammaf()` functions use the external integer `signgam` to return the sign of \( \Gamma(x) \), while `gamma_r()` and `gammaf_r()` use the user-allocated space addressed by `signgamp`.

The `signgam` variable isn’t set until `gamma()` or `gammaf()` returns. For example, don’t use the expression:

```c
float g = signgam * exp( gamma(x) );
```

to compute \( g = \Gamma(x) \). Instead, compute `gamma()` first:

```c
float lg = gamma(x);
float g = signgam * 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 `errno` to 0, call the function, and then check `errno` again. These functions don’t change `errno` if no errors occurred.
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>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.
getaddrinfo()  © 2006, QNX Software Systems GmbH & Co. KG.

The nodename and servname arguments are either pointers to null-terminated strings or NULL. One or both of these two arguments 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 (as well as setting ai_flags to 0)</td>
<td>hints</td>
<td>NULL</td>
</tr>
</tbody>
</table>

The hints argument defaults to all possibilities, but you can also use it to limit choices:
getaddrinfo()

- If the application handles only TCP but not UDP, you could set the `ai_socktype` member of the `hints` structure to `SOCK_STREAM`.
- 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.
getaddrinfo()

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.

**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/v6.

```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*/
```
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) {
    err1(1, cause);
    /*NOTREACHED*/
}
freeaddrinfo(res0);
```

```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) {
    err1(1, cause);
    /*NOTREACHED*/
}
freeaddrinfo(res0);
```
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>
</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:

addrinfo, freeaddrinfo(), gai_strerror()
getc() — Get the next character from a file

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;
```
getc()

```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>
</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:

cgetc() 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...*
Suddenly, the room filled with a heavy silence. Everyone seemed to be holding their breath, waiting for something to happen. The air was thick with anticipation, and the tension was palpable.

Someone cleared their throat, and the room fell silent once again. Everyone was on edge, waiting for the next move. The tension was so thick it seemed to be weighing down the room.

Finally, after what felt like forever, a voice broke the silence. It was a man's voice, deep and authoritative. He spoke slowly, each word carefully chosen. The room held its breath as they listened, their eyes fixed on the speaker.

"We need to act quickly," he said. "The situation is critical. We cannot afford to lose any more time."

The room's silence was broken by a chorus of agreement. Everyone knew that, at this point, any delay could mean disaster. They nodded, their взгляды focused on the speaker. He continued, his words resonating in the room:

"We have a plan. It's risky, but it's our best chance. We must act now."

The room fell silent once again, their minds racing with the possibilities and the outcomes. They knew they were facing a battle, but they were determined to win. The room was filled with a resolve that was almost palpable. They knew that, together, they could overcome anything.
Synopsis:

```c
#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).

Example:

```c
#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);
    }
}
```
fclose( fp );
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:

erno, feof(), ferror(), fgetc(), fgetchar(), getc(), putc(),
putc_unlocked(), putchar(), putchar_unlocked()
getchar_unlocked()
Get a character from stdin

Synopsis:
```
#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

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>
</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

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>
Caveats:

There is only one current working directory per process. In a multithreaded application, any thread calling \textit{chdir()} will change the current working directory for \textit{all threads} in that process.

See also:

\textit{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.
getdomainname()

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:

setdomainname()
getdtablesize()
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
```

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:

`close()`, `dup()`, `getrlimit()`, `open()`, `select()`, `sysconf()`
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

<table>
<thead>
<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:

geteuid(), getgid(), getuid(), setegid()
getenv()

Get the value of an environment variable

Synopsis:

```
#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:

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

int main( void )
{
    char* path;

    path = getenv( "INCLUDE" );
```
```c
if( path != NULL ) {
    printf( "INCLUDE=%s\n", path );
    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>No</td>
</tr>
</tbody>
</table>

**Caveats:**

The `getenv()` function manipulates the environment pointed to by the global `environ` variable.

**See also:**

`clearenv()`, `environ`, `execl()`, `execlp()`, `execlpe()`, `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

<table>
<thead>
<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:

getegid(), getgid(), getuid(), seteuid()
Synopsis:

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

gid_t getgid( void );
```

Library:

```c
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

<table>
<thead>
<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:

getegid(), geteuid(), getuid()
getgrent()

Return an entry from the group database

Synopsis:

```c
#include <grp.h>

struct group* getgrent( void );
```

Library:

```c
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()`
getgrent()  © 2006, QNX Software Systems GmbH & Co. KG.

* rewind()

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 ) {
        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>
</tbody>
</table>

continued…
Safety

Thread No

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.

\[\text{The getgrent()}, \text{getgrgid()}, \text{and getgrnam()} \text{functions share the same static buffer.}\]

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:

```
/*
 * Print a list of all users in your group
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <unistd.h>
```
```c
#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

**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:**

gettgr(), getgrgid_r(), getgrnam()
getgrgid_r()

Get information about the group with a given ID

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`
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 `getgrgid_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 `getgrgid_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>
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>
#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\n", argv[1] );
        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

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:

getgrent(), getgrgid(), getgrnam()
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 `struct 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`
getgrnam_r()

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 getgrnam_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 group structure.

The getgrnam_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>
</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:

getgrgid(), getgrgid_r(), getgrnam(), getlogin(), sysconf()
getgrouplist()  © 2006, QNX Software Systems GmbH & Co. KG.

Determine the group access list for a user

Synopsis:

#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: \n", 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

<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:

- 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:

- **gidsetsize**
  - The size of the `grouplist` array.
- **grouplist**
  - An array that the function can fill in with the process’s supplementary group IDs.

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:

- **EINVAL**
  - The `gidsetsize` argument isn’t equal to zero, and is less than the number of supplementary group IDs.
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( "%d\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()
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)\n",
        host ? host->h_name : "<unknown>",
        inet_ntoa( client.sin_addr ) );
```

874  C Library — F to H  November 2, 2006
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(), gethostent(), herror(), hostent, sethostent()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
gethostbyaddr_r()

Get a network host entry, in a thread-safe manner

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 `qcc` 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>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:

gethostbyaddr(), gethostbyname(), gethostbyname_r(), inet_ntop(), inet_pton()

/etc/hosts, /etc/resolv.conf in the Utilities Reference
Synopsis:

```
#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.
gethostbyname(), gethostbyname2() © 2006, QNX Software Systems GmbH & Co. KG.

If the name doesn’t contain a dot, and the environment variable 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 /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 use TCP and the connection is retained after each call to gethostbyname() or gethostbyaddr(). If the stayopen flag is zero, queries use UDP datagrams.

The gethostbyname2() function is an evolution of the 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 h_errno to NETDB_INTERNAL.

Returns:

A pointer to a valid hostent structure, or NULL if an error occurs (h_errno is set).

Errors:

See herror().

Files:

/etc/hosts Host database file.

/etc/resolv.conf Resolver configuration file.

For information about these files, see the Utilities Reference.

Environment variables:

HOSTALIASES

Name of the alias file that gethostbyname() is to search first when the hostname doesn’t contain a dot.
gethostbyname(),
gethostbyname2()

Classification:

gethostbyname() is POSIX 1003.1 OBS; gethostbyname2() is 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:

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(), gethostbyname2(), 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 bufflen,
    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.
- **bufflen** 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 `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>
</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(), gethostbyaddr_r(), gethostbyname()

/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>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>
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


**Synopsis:**

```c
#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.
**gethostent_r()**

**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_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, 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

<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:

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).
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>

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 `ifaddrs` 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:

```c
#include <unistd.h>

#define GETIOVBASE( _iov ) ...
```

Arguments:

`_iov` The iov_t structure from which you want to get the base member.

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>
<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:

GETIOVLEN(), SETIOV()
GETIOVLEN()  © 2006, QNX Software Systems GmbH & Co. KG.
Get the length member of an iov_t structure

Synopsis:
#include <unistd.h>
#define GETIOVLEN( _iov ) ...

Arguments:
    _iov    The iov_t structure from which you want to get the length member.

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** Decrement 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

**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:

alarm(), gettimeofday(), pthread_attr_setscope(), pthread_sigmask(),
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

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

continued...
**getlogin()**

© 2006, 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>No</td>
</tr>
</tbody>
</table>

**See also:**

`getlogin_r()`, `getpwnam()`, `getpwuid()`
getlogin_r()

Get the user name associated with the calling process

Synopsis:

```c
#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.

Remarque: 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.
getlogin_r()

Returns:

- **EOK**: Success.
- **ERANGE**: Insufficient storage was supplied via the name and namesize arguments to contain the user’s name.

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:

- 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.
getnameinfo()

© 2006, QNX Software Systems GmbH & Co. KG.

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 qcc 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>`:

```c
#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

**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:**

getaddrinfo(), gethostbyaddr(), getservbyport(), /etc/hosts,
/etc/resolv.conf, /etc/services, named
**getnetbyaddr()**

*Get a network entry, given an address (Unix)*

**Synopsis:**

```c
#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.
Files:

/etc/networks

Network name database file.

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>

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(), getnetbyname(), getnetent(), netent, setnetent()

/etc/networks in the Utilities Reference
**getnetbyname()**

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.
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>
<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>

914  C Library — F to H  November 2, 2006
See also:

`endnetent(), getnetbyaddr(), getnetbyname(), netent, setnetent()`

`/etc/networks` in the Utilities Reference
Parse options from a command line

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):

```plaintext
utility_name [-a] [-b] [-c option_argument]
             [-d] [-e] [-f option_argument] [operand ...]
```

Or:

```plaintext
utility_name [-ab] [-c option_argument]
             [-d] [-e] [-f option_argument] [operand ...]
```

Utilities with very complex arguments may be shown as:

```plaintext
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.
getopt()

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:
#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
Cancellation point Yes
Interrupt handler No

continued...
Safety

<table>
<thead>
<tr>
<th>Signal handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`getsubopt()`, `stderr`

getpass()  Prompt for and read a password

Synopsis:

```c
#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()**

Get the name of the peer connected to a socket

**Synopsis:**

```c
#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).

**Errors:**

- `EBADF` Invalid descriptor `s`. 
getpeername()

EFAULT     The name parameter points to memory not in a valid part of the process address space.
ENOBUFS    Insufficient resources were available in the system to perform the operation.
ENOTCONN   The socket isn’t connected.

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:

accept(), bind(), getsockname(), socket()
**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, \texttt{errno} is set to:

\begin{itemize}
\item ESRCH The process specified by \texttt{pid} doesn’t exist.
\end{itemize}

Classification:

POSIX 1003.1 XSI

\begin{center}
\begin{tabular}{l l}
\textbf{Safety} & \\
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}
\end{center}

See also:

\texttt{getsid()}, \texttt{setpgid()}, \texttt{setsid()}
Get the process group

Synopsis:

```c
#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:

```c
#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

Cancellation point  No
Interrupt handler No
Signal handler Yes
Thread Yes

See also:

setpgrp(), setsid()
**getpid()**

Get the process ID

**Synopsis:**

```c
#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:**

```c
#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

**Safety**

Cancellation point  No

*continued…*
getpid()

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:

getppid()
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()

<table>
<thead>
<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()
**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

<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 `getprio()` and `setprio()` functions are included in the QNX Neutrino libraries for porting QNX 4 applications. For new programs, use `sched_getparam()` or `pthread_getschedparam()`.

### See also:

`errno`, `pthread_getschedparam()`, `pthread_setschedparam()`, `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getparam()`, `sched_getscheduler()`, `sched_setscheduler()`, `sched_yield()`, `setprio()`
getprotobynumber()

Get a protocol entry, given a name

Synopsis:

```c
#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 `getprotobyname()`.

The `getprotobynumber()` and `getprotobyname()` 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

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>

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
getprotobynumber()  
Get a protocol entry, given a number

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.
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.

Currently, only the Internet protocols are understood.

See also:

endprotoent(), getprotobynumber(), getprotoent(), \texttt{protoent}, setprotoent()

/etc/protocols in the Utilities Reference
getprotoent()  © 2006, QNX Software Systems GmbH & Co. KG.

Read the next line of the protocol name database file

Synopsis:

```
#include <netdb.h>

struct protoent * getprotoent( void );
```

Library:

```
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>
**getprotoent()**

**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(), getprotobynumber(), protoent, setprotoent()

/etc/protocols in the Utilities Reference
getpwent()  © 2006, QNX Software Systems GmbH & Co. KG.
Get an entry from the password database

Synopsis:
#include <sys/types.h>
#include <pwd.h>

struct passwd* getpwent( void );

Library:
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.

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
/*
 * 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.
 */
#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>
```
```c
#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\n", argv[1] );
        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

<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:

getlogin(), getpwent(), getpwnam_r(), getpwuid()
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.
getpwnam_r()

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()

<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:**

getlogin(), getpwent(), getpwnam(), getpwuid(), getpwuid_r()
getpwuid() © 2006, QNX Software Systems GmbH & Co. KG.
Get information about the user with a given ID

Synopsis:
```
#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.

ExpThe 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:
```
/*
 * Print password info on the current user.
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
```
#include <pwd.h>

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(), getpwnam(), getpwuid(), getpwent()
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_GETPWR_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.
getpwuid_r()

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()

**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()`, `getpwnam()`, `getpwnam_r()`, `getpwuid()`
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()

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 built-in 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:

<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.
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(), execlp(), execvp(), execve(), execvp(), execvpe(), 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:
struct timeval ru_utime; /* user time used */
struct timeval ru_stime; /* system time used */
long ru_maxrss; /* max resident set size */
long ru_ixrss; /* integral shared memory size */
long ru_idrss; /* integral unshared data */
long ru_isrss; /* integral unshared stack */
long ru_miflnt; /* page reclaims */
long ru_majflt; /* page faults */
long ru_nswap; /* swaps */
long ru_inblock; /* block input operations */
long ru_oublock; /* block output operations */
long ru_msegd; /* messages sent */
long ru_msgrcv; /* messages received */
long ru_msignals; /* signals received */
long ru_mvcsw; /* voluntary context switches */
long ru_mivcsw; /* involuntary */

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_miflnt 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_outblock` 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_msgrcv` 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.
getrusage()

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_outblock` 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()`
gets() — Get a string of characters from standard input

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:

```
#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(), servant, 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.
getsid()

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:

errno, setsid()
getsockname()

Get the name of a socket

Synopsis:

```c
#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:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>Invalid descriptor s.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The name parameter points to memory that isn’t in a valid part of the process address space.</td>
</tr>
<tr>
<td>ENOBUFS</td>
<td>Insufficient resources were available in the system to perform the operation.</td>
</tr>
</tbody>
</table>

Classification:

POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Safety</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:

getpeername()
**getsockopt()**

Get options associated with a socket

**Synopsis:**

```c
#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_SNDBuffer, 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 `gcc` 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.
getsockopt()

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.
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:

(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.
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-tcpip.so` utility 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_SNDBUFFER**

*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()

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

<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

close(), getprotobynamel(), ioctl(), read(), select(), setsockopt(),
socket()

/etc/protocols in the Utilities Reference
Synopsis:

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

struct spwd* getspent( void );

struct spwd* getspent_r( struct spwd* result,
                          char* buffer,
                          int bufsize );
```

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];
    ```
getspent(), getspent_r()

setpwent( );
while( gets( buf ) != NULL ) {
    if( ( sp = getspnam( buf ) ) != ( struct spwd * )0 ) {
        printf( "Valid login name is: %s\n", sp->sp_namp );
    } else {
        setspent( );
        while( ( sp=getspent( ) ) != ( struct spwd * )0 )
            printf( "%s\n", 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>
See also:

errno, fgetspent(), getgrent(), getlogin(), getspnam(), getpwuid(),
putspent(), setspent()
getspnam(), getspnam_r()
Get information about a user with a given name

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 `fgetspent()`, `getspent()`, and `getspnam()` functions share the same static buffer.

Returns:

A pointer to an object of type `struct spwd` containing an entry from the group database with a matching 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("%s username \n", argv[0]);
        return(EXIT_FAILURE);
    }
    
    sp = getspnam( argv[1] );
    if ( (sp == (struct spwd*)0) ) {
        fprintf( stderr, "getspnam: unknown %s\n", argv[1] );
        return( EXIT_FAILURE );
    }
    
    printf( "login name %s\n", sp->sp_namp );
    printf( "password %s\n", sp->sp_pwdp );
    return( EXIT_SUCCESS );
}
```
Classification:

Unix

getspnam()

<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>

getspnam_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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

fgetspent(), getlogin(), getspent(), getpwuid(), putspent()
**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()` function 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: **rw**, **hard**, **bg**, and **wsize**, the last of which has an associated value of 1024.

The `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 `optionp` contains only one suboption, `getsubopt()` updates `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 `optionp` to point to the start of the next suboption. If the suboption has an associated value, `getsubopt()` updates `valuep` to point to the value’s first character. Otherwise, it sets `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 `getsubopt()` returns, if `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 `getsubopt()` fails to match the suboption with the tokens in the `tokens` array, the calling program should decide if this is an error, or if the unrecognized option should be passed to another program.

**Returns:**

The `getsubopt()` function returns -1 when the token it’s scanning isn’t in the `tokens` vector. The variable addressed by `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 `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;
                ```
getsubopt()

```c
    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**

<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…
getsubopt()

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:

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:

g getopt()
gettimeofday()

Get the current time

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()  

Errors:

EFAULT  An error occurred while accessing the when buffer.

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:

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(), clock_gettime(), clock_settime(), ctime(), ctime(), difftime(), gmtime(), gmtime(), localtime(), localtime(), settimeofday(), time()
getuid()  
Get the user ID

Synopsis:

```c
#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:

```c
#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
getuid()

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(), geteuid(), getgid(), setuid()


**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:**

`_PATH_UTMP`

Specifies the user information file.

**Classification:**

Unix
getutent()

<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 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.
getutid()

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

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>

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 `utline` string that matches `line->utline`. 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(), getutental, getutid(), pututline(), setutent(), utmp, utmpname()

login in the Utilities Reference
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

<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:

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()`, `ungetc_unlocked()`.
getwc()

Read a wide character from a stream

Synopsis:

```
#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).

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 `fp` stream isn’t valid for reading.
- `EINTR` A signal terminated the read operation; no data was transferred.
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

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, feof(), ferror(), putwc(), putwchar()

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter
**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()

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:
#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 (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:**

```c
#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.
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 `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->gloffs to specify how many NULL pointers to add at the beginning of pglob->pathv. After the call to glob(), pglob->pathv will contain pglob->gloffs NULL pointers, followed by pglob->glpathc 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 errfunc 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
", idx,
                     paths.gl_pathv[idx] );
        }
    else {
            puts( "glob() failed" );
    }

    return 0;
}
```
glob() 

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:

Don’t change the values in pglob between calling glob() and globfree().

See also:

globfree(), wordexp(), wordfree()
**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

<table>
<thead>
<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>
globfree()

Caveats:

Don’t change the values in `pglob` between calling `glob()` and `globfree()`.

See also:

`glob()`, `wordexp()`, `wordfree()`
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 \texttt{tm} structure that contains the broken-down time.

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:

\texttt{asctime()}, \texttt{asctime()}\texttt{r()}, \texttt{clock()}, \texttt{ctime()}, \texttt{difftime()}, \texttt{gmtime()}\texttt{r()}, \texttt{localtime()}, \texttt{localtime()}\texttt{r()}, \texttt{mktime()}, \texttt{strftime()}, \texttt{time()}, \texttt{tm}, \texttt{tzset()}

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino \textit{User’s Guide}
**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 \texttt{tm} structure containing the broken-down time.

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:

\texttt{asctime()}, \texttt{asctime_r()}, \texttt{clock()}, \texttt{ctime()}, \texttt{difftime()}, \texttt{localtime()}, \texttt{localtime_r()}, \texttt{mktime()}, \texttt{strftime()}, \texttt{time()}, \texttt{tm}, \texttt{tzset()}

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino \textit{User’s Guide}
**Synopsis:**

```c
#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()`, `gethostname()`, `gethostname2()`, `gethostname_r()`, `res_query()`, `res_search()`
**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

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:

bsearch(), hdestroy(), hsearch(), malloc()

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
See also:

hcreate(), hsearch()
**Synopsis:**

```
#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()`
- `gethostbyname2_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:
herror()

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:

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(), gethostbyaddr_r(), gethostbyname(),
gethostbyname_r(), h_errno, res_query(), res_search(), stderr
hostent
Structure that describes an Internet host

Synopsis:
```c
#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;
    ENTRY item, *found_item;
    /* name to look for in table */
    char name_to_find[30];
    int i = 0;

    /* create table */
    (void) hcreate(NUM_EMPL);

    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
                 &info_ptr->room) != EOF && i++ < NUM_EMPL) {
```
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",
            found_item->key,
            (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>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:

bsearch(), hcreate(), hdestroy(), malloc(), strcmp()

The Art of Computer Programming, Volume 3, Sorting and Searching
Synopsis:

```c
#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>
</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:**

*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()
**htons()**

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

**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

<table>
<thead>
<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:

```
#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()**

© 2006, 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

<table>
<thead>
<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>

1052 C Library — F to H

November 2, 2006
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*
**hwi_find_tag()**

Find a tag in the hwinfo structure

**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).
**hwi_find_tag()**

**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>
<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:**

`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.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**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

  `continued...`
hwi_off2tag()

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:

hwi_find_item(), hwi_find_tag(), hwi_tag2off()

“Structure of the system page” in the Customizing Image Startup Programs chapter of Building Embedded Systems
hwi_tag2off()

Return the offset from the start of the hwinfo area of the system page

Synopsis:

```c
#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.

Note: 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

Cancellation point No

continued...
hwi_tag2off()

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:

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:

```
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

Safety

Cancellation point  No
Interrupt handler   No
Signal handler     No
Thread             Yes
See also:

\textit{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>l trunc()</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>
ICMP

Internet Control Message Protocol

Synopsis:
#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.

ENOBUPS 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(), getprotobynumber(), read(), recv(), recvfrom(), send(), sendto(), socket(), write()

RFC 792
ICMP6

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

```
```
ICMP6

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(), getprotobynamel(), getsockopt(), read(), recv(), recvfrom(),
send(), sendto(), setsockopt(), socket(), write()

RFC 2292
Synopsis:
#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>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:
getifaddrs(), if_indextoname(), if_nameindex(), if_nametoindex()
**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

**Safety**

- Cancellation point : Yes
  
  *continued...*
**if_indextoname()**

© 2006, QNX Software Systems GmbH & Co. KG.

<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:**

getifaddrs(), if_freenameindex(), if_nameindex(), if_nametoindex()
if_nameindex()

Return a list of interfaces

Synopsis:

```
#include <net/if.h>

struct if_nameindex * if_nameindex( void );
```

Library:

```
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
### 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:**

`getifaddrs()`, `if_freenamelnex()`, `if_indextoname()`, `if_nametoinde()`
Synopsis:

```
#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

<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:

getifaddrs(), if_freenameindex(), if_indextoname(), if_nameindex()
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 `<ifaddr.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>
**ilogb(), ilogbf()**

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() {
  printf( "%f\n", ilogb(.5) );
  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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`log(), logb(), log10(), log1p()`
Synopsis:
#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

    Safety

    Cancellation point    No
    Interrupt handler     Yes
    Signal handler        Yes
    Thread                Yes
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()`
in8s()
Read 8-bit values from a port

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>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(), in16(), in16s(), in32(), in32s(), mmap_device_io(), 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**

<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.

Both `inbe16()` and `inle16()` are implemented as macros.

**See also:**

`in8()`, `in8s()`, `in16()`, `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
**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_IOCTL` 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
**in32(), inbe32(), inle32()**

© 2006, 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>

### 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
in32s()

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.

See also:

in8(), in8s(), in16(), in16s(), in32(), mmap_device_io(), out8(),
out8s(), out16(), out16s(), out32(), out32s()
Synopsis:

```
#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...
### 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:**

rindex(), strchr(), strrchr()
inet_addr()
Convert a string into a numeric Internet address

Synopsis:

#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**

<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:**

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()`
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()**

© 2006, 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_ifaddrs()`, `inet_makeaddr()`, `inet_netof()`, `inet_network()`, `inet_ntoa()`,

`/etc/hosts`, `/etc/networks` in the *Utilities Reference*
Synopsis:

```c
#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

<table>
<thead>
<tr>
<th>Safety</th>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>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()`
inet_makeaddr()
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()**

© 2006, 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:**

*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:

\textit{a.b.c.d/bits} or \textit{a.b.c.d}

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 “\textit{d.c.b.a}”.

\textit{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 \textit{net.net.host}.

\textit{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 \textit{net.host}. 
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>
</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_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_ntop()`.

**Returns:**

An Internet network number.

**Classification:**

QNX Neutrino

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
</tbody>
</table>

*continued...*
See also:

*inet_aton()*, *inet_lnaof()*
**inet_net_pton()**

Convert an Internet network number from CIDR format to network format

**Synopsis:**

```c
#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:**

```c
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()`.
**inet_netpton()**

**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>
<th></th>
</tr>
</thead>
<tbody>
<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_network()**

*Convert a string into an Internet network number*

**Synopsis:**

```c
#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**

<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:**

`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_ntop()`.

Returns:

A string representing an Internet address.

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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<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_ntoa()
**inet_ntop()**

Convert a numeric network address to a string

**Synopsis:**
```c
#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`.

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:

```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_ntop(AF_INET, INADDR, &inaddr)) == 0) {
        printf("Invalid address: %s\n", INADDR);
        exit(EXIT_FAILURE);
    } else if (rval == -1) {
        perror("inet_ntop");
        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>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:

inet_ntop()
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:
### Family Numeric address size

<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_aton()` 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:
  
  \[ x:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx \]
  
  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:

  \[
  1080:0:0:0:8:800:200C:417A
  \]

  \[
  FF01:0:0:0:0:0:0:43
  \]

  \[
  0:0:0:0:0:0:0:1
  \]

  \[
  0:0:0:0:0:0:0:0
  \]

  can be represented as:
inet_pton()

A convenient format when dealing with mixed IPv4 and IPv6 environments is:

\[ x:x:x:x:x:d.d.d.d \]

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:

\[
\begin{align*}
0:0:0:0:0:0:13.1.68.3 \\
0:0:0:0:0:FFFF:129.144.52.38
\end{align*}
\]

Or, in their compressed forms:

\[
\begin{align*}
::13.1.68.3 \\
::FFFF:129.144.52.38
\end{align*}
\]

Returns:

- 1 Success.
- 0 The input isn’t a valid address.
- -1 An error occurred (\textit{errno} is set).

Errors:

- EAFNOSUPPORT

The \( af \) 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);
}
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:

*inet_ntop()*

*RFC 2373*
Synopsis:

```
#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 <netinet6/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
Synopsis:

```c
#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 \( x \times n + y \). It must have a value of 1, 2, 4, or 8.

- `plusy` Value y in the alignment term \( x \times n + y \). It must have a value between 0 and 7, inclusive.

Library:

**libsocket**

Use the `-l socket` option to **qcc** to link against this library.
**inet6_option_alloc()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_append(), inet6_option_find(), inet6_option_init(), inet6_option_next(), inet6_option_space()*


**inet6_option_append()**

Append an IPv6 hop-by-hop or destination option to an ancillary data object.

**Synopsis:**

```c
#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.
**inet6_option_append()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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

<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_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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_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

<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:

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 of 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.
inet6_option_next()

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()


S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6)
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 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()


**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

<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_getaddr(), inet6_rthdr_getflags(), inet6_rthdr_init(), inet6_rthdr_lasthop(), inet6_rthdr_reverse(), inet6_rthdr_segments(), inet6_rthdr_space()*


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
**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:**

- `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:

```c
#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.
**inet6_rthdr_getflags()**

**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_init(),
inet6_rthdr_lasthop(), inet6_rthdr_reverse(), inet6_rthdr_segments(),
inet6_rthdr_space()


**inet6_rthdr_init()**

Initialize an IPv6 routing header

**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

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_lasthop()`, `inet6_rthdr_reverse()`, `inet6_rthdr_segments()`, `inet6_rthdr_space()`


Synopsis:

```c
#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.
Classificação:

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>

Veja também:

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

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_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

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>continued . .</td>
<td></td>
</tr>
</tbody>
</table>

November 2, 2006
**inet6_rthdr_segments()**

<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:**

`inet6_rthdr_add()`, `inet6_rthdr_getaddr()`, `inet6_rthdr_getflags()`, `inet6_rthdr_init()`, `inet6_rthdr_lasthop()`, `inet6_rthdr_reverse()`, `inet6_rthdr_space()`


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

<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_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>
<th></th>
</tr>
</thead>
<tbody>
<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:

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:

getgroups(), getgrnam()
### 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.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

### 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:

**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\n", 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:

#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

<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>
InterruptAttach(), InterruptAttach_r()

Attach an interrupt handler to an interrupt source

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.
**InterruptAttach()**, **InterruptAttach_r()** © 2006, QNX Software Systems GmbH & Co. KG.

**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

```
$(QNX_TARGET)/$(PROCESSOR)/boot/build.
```
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 intr</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 . . .
InterruptAttach(), InterruptAttach_r() © 2006, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Interrupt intr</th>
<th>Description</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.
InterruptAttach()
InterruptAttach_r() 

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.

 НТО_INTR_FLAGS_PROCESS

Adding НТО_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.

 НТО_INTR_FLAGS_TRK_MSK

The НТО_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 НТО_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:

- **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 I/O privileges.

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:**

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*
**Synopsis:**

```
#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.
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 `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_SIGNAL_THREAD 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()** ©

2006, QNX Software Systems GmbH & Co. KG.

<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 <code>InterruptMask()</code> and <code>InterruptUnmask()</code> 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).
 interrupt Attach Event() ,
interrupt Attach Event_r()

_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 calls 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

_interrupt Attach Event() 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.

Blocking states

This call doesn’t block.
InterruptAttachEvent(), InterruptAttachEvent_r()

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

<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>

Copyright © 2006, QNX Software Systems GmbH & Co. KG.
See also:

`InterruptAttach()`, `InterruptDetach()`, `InterruptLock()`, `InterruptMask()`, `InterruptUnlock()`, `InterruptUnmask()`, `InterruptWait()`, `sigevent`

Writing an Interrupt Handler chapter of the Neutrino *Programmer’s Guide*
InterruptDetach(), InterruptDetach_r()

Detach an interrupt handler by ID

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

<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(), InterruptAttachEvent(), InterruptHookIdle(), InterruptUnlock()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
InterruptDisable()

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:

```c
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.
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

<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:

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
InterruptHookIdle()

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><code>_NTO_INTR_FLAGS_END</code></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><code>_NTO_INTR_FLAGS_PROCESS</code></td>
<td>Associate the handler with the process instead of the attaching thread.</td>
</tr>
<tr>
<td><code>_NTO_INTR_FLAGS_TRK_MSK</code></td>
<td>Track calls to <code>InterruptMask()</code> and <code>InterruptUnmask()</code> 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 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.
InterruptHookIdle()

_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 unmask operations 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()

Synopsis:

#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
Synopsis:

```c
#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:

```c
memset( spinlock, 0, sizeof( *spinlock ) );
```

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:

```c
ThreadCtl( _NT0_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:

```c
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

<table>
<thead>
<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>
InterruptLock()

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:

```
#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:

```
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.

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

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>

### 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
InterruptUnlock()

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(), InterruptLock(), InterruptMask(), InterruptUnmask(), ThreadCtl()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
**Synopsis:**

```
#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:**

`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:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

If the thread doesn’t do this, it might SIGSEGV when it calls `InterruptUnmask()`.
InterruptUnmask()  

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 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>
<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:

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.
InterruptWait(), InterruptWait_r() © 2006, QNX Software Systems GmbH & Co. KG.

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.

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:

EINTR The call was interrupted by a signal.
ENOTSUP The reserved arguments aren’t NULL.
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:**

InterruptAttach(), InterruptAttachEvent(), TimerTimeout()

Writing an Interrupt Handler chapter of the Neutrino Programmer’s Guide
**_intr_v86()_**

Execute a real-mode software interrupt

**Synopsis:**

```
#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:
Interrupt Description

int 10h  Video BIOS
int 1ah  PCI

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 datasync 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
", reg.eax, reg ebx, reg ecx, reg edx);
    printf("esi=%8x edi=%8x ebp=%8x esp=%8x
", reg esi, reg edi, reg ebp, reg esp);
    printf(" ds=%8x es=%8x fs=%8x gs=%8x
", reg ds, reg es, reg fs, reg gs);
    printf("efl=%8x\n\n", reg.efl);
}

Classification:

QNX Neutrino (x86 only)

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>
Structure of a resource manager’s connect message

Synopsis:

```c
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

- _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.
**mode**

Contains 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_IRWXO</td>
<td>Read, write, execute/search. A bitwise inclusive OR of the other three constants (S_IRWXU is OR of IROSR, 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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IEXEC</td>
<td>S_IIXUSR</td>
</tr>
<tr>
<td>S_IREAD</td>
<td>S_IIRUSR</td>
</tr>
</tbody>
</table>

*continued...*
**_io_connect_**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<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
- _IO_CONNECT_EXTRA_PHOTON
- _IO_CONNECT_EXTRA_SOCKET
• _IO_CONNECT_EXTRA_SEM
• _IO_CONNECT_EXTRA_RESMGR_LINK
• _IO_CONNECT_EXTRA_PROC_SYMLINK
• _IO_CONNECT_EXTRA_RENAME
• _IO_CONNECT_EXTRA_MOUNT
• _IO_CONNECT_EXTRA_MOUNT_OCB

extra_len  The length of any extra data included in the message.

path  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.
  - `_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.
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()**

*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 c` 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.
**ioctl()**

**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

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: No

**Caveats:**

The *ioctl()* function is a Unix function that varies greatly from platform to platform.

**See also:**

- *devctl()*
iofdinfo() 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`. 
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()`
**Synopsis:**

```c
#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 `-l c` 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

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_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:

```c
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.
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:

`iofunc_attr_init()`, `iofunc_attr_t`, `iofunc_attr_trylock()`, `iofunc_attr_unlock()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

November 2, 2006
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;
  list_size;
#if !_defined(_IOFUNC_OFFSET_BITS) || _IOFUNC_OFFSET_BITS == 64
#if _FILE_OFFSET_BITS == 64
  off_t nbytes;
  ino_t inode;
#else
  off64_t nbytes;
  ino64_t inode;
#endif
#else
#if defined(_LITTLEENDIAN)
  off_t nbytes;
  off_t nbytes_hi;
  ino_t inode;
  ino_t inode_hi;
#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
```

```
The `iofunc_attr_t` structure describes the attributes of the device that’s associated with a resource manager. The members include the following:

**Description:**

A pointer a structure 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_*)..

`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()`.

A thread must unlock the attributes as many times as it locked them.

`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.
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.

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.

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()).

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*. 
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.
**iofunc_attr_trylock()**

**Returns:**

- **EOK**  Success.
- **EBUSY**  The calling thread couldn’t lock the attributes immediately.
- **EAGAIN**  On the first use, all kernel mutex objects were in use.

**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:**

- `iofunc_attr_init()`, `iofunc_attr_lock()`, `iofunc_attr_unlock()`, `iofunc_attr_unlock()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Unlock the attribute structure

Synopsis:

```
#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
See also:

`iofunc_attr_init()`, `iofunc_attr_lock()`, `iofunc_attr_unlock()`, `iofunc_attr_trylock()

Writing a Resource Manager chapter of the *Programmer’s Guide*.


**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

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_client_info(), iofunc_open(), iofunc_read_verify(),
iofunc_write_verify()

Writing a Resource Manager chapter of the Programmer’s Guide.
Handle an IO_CHMOD message

Synopsis:

```
#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_chgmod()` 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_chgmod()`, 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_chgmod_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;
};

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_t` 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
iofunc_chmod()

information, see “Combine messages” in the Writing a Resource Manager chapter of the Programmer’s Guide.

mode

The new mode. For more information, see “Access permissions” in the documentation for stat().

Returns:

EOK Successful completion.
EROMS 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

<table>
<thead>
<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_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:

```
l libc
```

Use the `-l c` option to `gcc` 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_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
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_TIME, 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 IOCHOWN 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** `IOCHOWN`.
- **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.
### iofunc\_chown()

#### 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\_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*. 

---

© 2006, QNX Software Systems GmbH & Co. KG.
**iofunc_chown_default()**  © 2006, QNX Software Systems GmbH & Co. KG.

Default handler for _IO_CHOWN_ messages

**Synopsis:**

```c
#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.
- **EROSFS** 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_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*. 
**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.
EINVVAL The client process is no longer valid.
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:

ConnectClientInfo()

Writing a Resource Manager chapter of the Programmer’s Guide.
**Synopsis:**

```c
#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.
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;
};

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


**iofunc_close_dup()**

© 2006, 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:**

- `iofunc_close_dup_default()
- `iofunc_close_ocb()`

Writing a Resource Manager chapter of the *Programmer’s Guide.*
Default handler for IO_CLOSE messages

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>
</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_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:

```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() function detaches the OCB specified by `ocb`, and releases the memory associated with it.
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 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_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:

- `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>
</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_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:

- `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]; */
};
```

```c
struct _io_devctl_reply {
    uint32_t zero;
};
```
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 \textit{i} member is a structure of type \texttt{io_devctl} that contains the following members:

\begin{itemize}
  \item \texttt{type} \texttt{IO_DEVCTL}.
  \item \texttt{combine_len} If the message is a combine message, \texttt{IO_COMBINE_FLAG} is set in this member. For more information, see “Combine messages” in the Writing a Resource Manager chapter of the \textit{Programmer’s Guide}.
  \item \texttt{dcmd} The device-control command to execute.
  \item \texttt{nbytes} The number of bytes of data being passed with the command.
\end{itemize}

The commented-out declaration for \textit{data} indicates that \texttt{nbytes} bytes of data immediately follow the \texttt{io_devctl} structure.

The \texttt{_DEVCTL_DATA()} macro gets a pointer to the data that follows the message. Call it like this:

\begin{verbatim}
data = _DEVCTL_DATA (msg->i);
\end{verbatim}

The \texttt{o} member of the \texttt{io_devctl_t} message is a structure of type \texttt{io_devctl_reply} that contains the following members:
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

<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:

- fcntl(), iofunc_attr_t, iofunc_devctl_default(), iofunc_ocb_t, resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
iofunc_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.
EINV    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

<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_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:

```
#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()`.

**Library:**

```text
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:

`ioctlinfo()`, `iofunc_attr_t`, `iofunc_fdinfo_default()`,
`iofunc_ocb_t`, `resmgr_context_t`, `resmgr_pathname()

Writing a Resource Manager chapter of the *Programmer’s Guide*.
iofunc_fdinfo_default()  © 2006, QNX Software Systems GmbH & Co. KG.
Default handler for _IO_FDIINFO messages

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_FDIINFO` message by the resource manager.

You can place this function directly into the `io funcs` table passed to `resmgrAttach()`, 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.
The `io_fdinfo_t` structure holds the `IO_FDINFO` 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;
};
```

```c
struct io_fdinfo_reply {
    uint32_t zero[2];
    struct fdinfo info;
    /* char path[path_len + 1]; */
};
```

```c
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_FDINFO`.
- **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

<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:

`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()**

<table>
<thead>
<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()`, `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 `_client_info` structure that contains information about the client. For information about this structure, see `ConnectClientInfo()`.

Library:

`libc`

Use the `-l` 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
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.
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:

```c
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_lock_calloc()`, `iofunc_lock_free()`

Writing a Resource Manager chapter of the *Programmer’s Guide*.
Synopsis:

```c
#include <sys/iofunc.h>

iofunc_lock_list_t *iofunc_lockcalloc
    ( 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 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_lockcalloc()` 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.
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*. 

1290 C Library — I to L  November 2, 2006
Synopsis:
#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
iofunc_lock_default() (© 2006, QNX Software Systems GmbH & Co. KG.)

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>
</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_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*. 
**iofunc_lock_free()**

Return memory allocated for lock structures

**Synopsis:**

```c
#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>

© 2006, QNX Software Systems GmbH & Co. KG.
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 c` 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_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` 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.
The function `iofunc_lseek()` is a system call used in the QNX Neutrino environment. It allows for the manipulation of the position within a file. The function takes the following parameters:

- `whence`: SEEK_SET, SEEK_CUR, or SEEK_END.
- `offset`: The relative offset from the file position determined by the `whence` member.

The `offset` 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>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_attr_t`, `iofunc_lseek_default()`, `iofunc_ocb_t`, `lseek()`, `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

November 2, 2006
Synopsis:

```c
#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.
iofunc_lseek_default() © 2006, QNX Software Systems GmbH & Co. KG.

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

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**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*. 

1302 C Library — I to L November 2, 2006
**iofunc_mknod()**

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

**Synopsis:**

```c
#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.
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.

```c
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 dattr.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the info buffer.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The client process is no longer valid.</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>NULL was passed in info.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The group ID or owner ID didn’t match.</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:

- _io_connect, _io_connect_link_reply, _io_connect_f_type_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:**

```
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 \texttt{iofunc\_mmap()} function.

\textbf{\texttt{io\_mmap\_t structure}}

The \texttt{io\_mmap\_t} structure holds the \_IO\_MMAP message received by the resource manager:

\begin{verbatim}
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;
\end{verbatim}

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 \texttt{i} member is a structure of type \_io\_mmap that contains the following members:

\begin{description}
\item[type] \_IO\_MMAP.
\item[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 \textit{Programmer’s Guide}.
\end{description}
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.

The offset into shared memory of the location that the client wants to start mapping.

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. ≤ 0)
  
  Successful completion.

- EROFS An attempt to memory map (mmap) a read-only file, using the PROT_WRITE page protection mode.
The client doesn’t have the appropriate permissions.

Insufficient memory exists to allocate internal resources required to effect the mapping.

**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_mmap_default(), iofunc_ocb_t, 
_msg_info, resmgr_context_t

Writing a Resource Manager chapter of the *Programmer’s Guide.*
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 c` 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. \( \leq 0 \))

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>
</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_mmap(), 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_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:

\[
\begin{align*}
_{\text{NOTIFY\_COND\_INPUT}} &\equiv _{\text{NOTIFY\_CONDE\_RDNORM}} \\
_{\text{NOTIFY\_COND\_OUTPUT}} &\equiv _{\text{NOTIFY\_CONDE\_WRNORM}} \\
_{\text{NOTIFY\_COND\_OBAND}} &\equiv _{\text{NOTIFY\_CONDE\_RDBAND}}
\end{align*}
\]

Setting the _NOTIFY\_COND\_EXTEN flag affects the 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 nop array in order for the event to be triggered. Both this array and the 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 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 trig argument, and armed isn’t NULL, then when you call iofunc_notify(), armed must contain the number of elements in the nop and notifycounts arrays (provided notifycounts isn’t NULL). Otherwise, the function assumes there are three elements in the nop and notifycounts arrays.
**iofunc_notify()**

**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_POLLARM**
  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:
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() — 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_OUTPUT — the condition is met when one or more units of out-of-band data are available.

- NOTIFYCOND_OBAND — the condition is met when one or more units of out-of-band data are available.

- NOTIFYCOND_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 _io_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.
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_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
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_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:

```c
#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 << 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*.
iofunc_ocb_attach()  © 2006, QNX Software Systems GmbH & Co. KG.
Initialize an Open Control Block

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.
iofunc_ocb_attach()

Description:

The `iofunc_ocb_attach()` function examines the mode specified by the `io_open` 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.

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

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_t`, `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*. 

November 2, 2006  C Library — I to L  1325
Allocate an iofunc Open Control Block

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_ocbcalloc()` 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_ocb_calloc()` 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_ocb_calloc()` and `iofunc_ocb_free()` callouts in the attribute’s mount structure, then you should use the callouts instead of calling the standard `iofunc_ocb_calloc()` and `iofunc_ocb_free()` functions.

**Returns:**

A pointer to an `iofunc_ocb_t` OCB structure.

**Examples:**

Override `iofunc_ocb_calloc()` and `iofunc_ocb_free()` to manage an extended OCB:

```c
typedef struct {
    iofunc_ocb_t iofuncOCB; /* the OCB used by iofunc_* */
    int myFlags;
    char moreOfMyStuff;
} MyOCBT;

MyOCBT *iofunc_ocb_calloc (resmgr_context_t *ctp,
                           iofunc_attr_t *attr)
{
    return ((MyOCBT *) calloc (1, sizeof (MyOCBT)));
}

void iofunc_ocb_free (MyOCBT *ocb)
{
    free (ocb);
}
```
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*. 
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:

```c
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.
iofunc_ocb_detach()

Classification:

QNX Neutrino

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:

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 (`scaloc()` 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:

```c
#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.
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);

return(_RESMGR_PTR(ctp, &msg->link_reply, len));
}

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:

/*
* 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_ocbAttach
return EOK

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_attr_init(),
iofunc_open()

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:

```c
#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()` to check the client’s open mode against the resources attributes to see if the client can open the resource in that mode

• `iofunc_ocb_calloc()` to allocate an Open Control Block (OCB)

• `iofunc_ocb_attach()` to initialize the OCB

• `resmgr_open_bind()` 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_func_init()` 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_open_bind()`.

**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_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*. 
iofunc_openfd() Increment count and locking flags

Synopsis:

```c
#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.
iofunc_openfd()

- **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 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>
</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_openfd()`, `iofunc_openfd_default()`, `.msg_info`, `openfd()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Default handler for IO_OPENFD messages

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EACCES</td>
<td>You don’t have permission to open the file.</td>
</tr>
<tr>
<td>EBUSY</td>
<td>The file has shared locks that are in use.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The message type is invalid.</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:

`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.*
**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_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>
</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_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()`. The `iofunc_pathconf_default()` function simply calls `iofunc_pathconf()` to do the actual work.

**Returns:**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The pathconf parameter being ascertained wasn’t one of _PC_CHOWN_RESTRICTED, _PC_NO_TRUNC, or _PC_SYNC_IO.</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:**

`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*. 
iofunc_read_default()
Default handler for IO_READ messages

Synopsis:

```c
#include <sys/iofunc.h>

int iofunc_read_default( resmgr_context_t *ctp,
                         io_read_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_read_t* structure that contains the message that the resource manager received. For more information, see the documentation for *iofunc_read_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_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 *iofunc_func_init()* to initialize all of the functions to their default values.

The *iofunc_read_default()* function calls *iofunc_read_verify()* to do the actual work.
Returns:

EBADF   The client doesn’t have read access to this resource.
EINV    The extended type information is invalid.
EOK     The client has read access to this 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_read_verify()`,
`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_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:

```c
_IO_SET_READ_NBYTES( resmgr_context_t *ctx,
            int nbytes )
```

_io_read_t structure

The `io_read_t` structure holds the _IO_READ message received by the resource manager:

```c
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:

- **\( \text{type} \)** `_IO_READ`
- **\( \text{combine_len} \)** 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*.
- **\( \text{nbytes} \)** The number of bytes that the client wants to read.
- **\( \text{xtype} \)** 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

<table>
<thead>
<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_verify()`

Writing a Resource Manager chapter of the *Programmer’s Guide*
**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:

```c
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

<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:

_{io_connect, io_connect_link_reply, io_connect_ftype_reply, iofunc_open, readlink}_

Writing a Resource Manager chapter of the _Programmer's Guide_.

November 2, 2006
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().
Library:

libc

Use the `-l c` 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:

```c
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:
• \texttt{io_connect_ftype_reply} if the reply consists of a status and a file type.

The reply includes the following extra information:

\begin{verbatim}
typedef union _io_rename_extra {
    char path[1];
} io_rename_extra_t;
\end{verbatim}

\section*{Returns:}

\begin{itemize}
    \item \textbf{EACCES} The client doesn’t have permissions to do the operation.
    \item \textbf{EBADFSYS} NULL was passed in \textit{oldattr}, \textit{olddattr}, or \textit{newdattr}.
    \item \textbf{EFAULT} A fault occurred when the kernel tried to access the \textit{info} buffer.
    \item \textbf{EINVAL} The \textit{oldattr} and \textit{newdattr} have identical values, the client process is no longer valid, or attempt to remove the parent (".") directory.
    \item \textbf{EISDIR} The old link is a directory but the new link isn’t a directory.
    \item \textbf{ENOTDIR} Attempt to unlink a nondirectory entry using directory semantics (e.g. \texttt{rmdir file}).
    \item \textbf{ENOTEMPTY} Attempt to remove a directory that isn’t empty.
    \item \textbf{EOK} Successful completion or there was already a \textit{newattr} entry.
    \item \textbf{EPERM} The group ID or owner ID didn’t match.
    \item \textbf{EROFS} Attempt to remove an entry on a read-only filesystem.
\end{itemize}
iofunc_rename()

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()

Writing a Resource Manager chapter of the Programmer's Guide.
iofunc_space_verify()  
© 2006, QNX Software Systems GmbH & Co. KG.

Do permission checks for the IO SPACE message

Synopsis:

```c
#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**

<table>
<thead>
<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_t`
- `iofunc_open()`, `iofunc_write_default()`, `iofunc_write_verify()`, `resmgr_context_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
iofunc_stat()  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>
</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_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:**

```c
#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.

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:

```c
#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.*
Synopsis:

```
#include <sys/iofunc.h>

int iofunc_sync_default( resmgr_context_t *ctp,
                        io_sync_t *msg,
                        iofunc_ocb_t *ocb );
```

Arguments:

cptr 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:

- **EINV AL** 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:

```c
#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:

```c
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:

```c
struct _io_sync {
    uint16_t type;
    uint16_t combine_len;
};
```
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:**

- EINV\_AL The resource doesn’t support syncing.
- EOK The client has read access to this resource.
iofunc_sync_verify()

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_ocb_t, iofunc_open(), iofunc_write_default(), iofunc_write_verify(), resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
**Synopsis:**

```c
#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:**

- `libc`

  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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
**iofunc_unblock()**

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`. 
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*. 

November 2, 2006
iofunc_unblock_default() © 2006, QNX Software Systems GmbH & Co. KG.

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 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>
</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_func_lock(), 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_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` 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.
iofunc_unlink()

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.

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, ConnectClientInfo(), iofunc_attr_t, iofunc_check_access(), resmgr_context_t

Writing a Resource Manager chapter of the Programmer’s Guide.
**iofunc_unlock_ocb_default()** © 2006, QNX Software Systems GmbH & Co. KG.

Default handler for the unlock_ocb callout

**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:**

`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 `iofuncs` 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_unlock(), iofunc_func_init(), iofunc_ocb_t, resmgrAttach(), 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_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 JO_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** `JO_UTIME`
- **combine_len** If the message is a combine message, `JO_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.
- **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

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_time_update()`, `iofunc_utime_default()`, `utime()`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 
Synopsis:

```c
#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 c` option to `gcc` 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 `ocb->attr` structure. This function takes care of updating these bits in the flags member of `ocb->attr`:

- IOFUNC_ATTR_ATIME
- IOFUNC_ATTR_CTIME
- IOFUNC_ATTR_MTIME
- IOFUNC_ATTR_DIRTY_TIME
- IOFUNC_ATTR_DIRTY_MODE

The `iofunc_utime()` function then calls `iofunc_time_update()` to update the appropriate time fields in `ocb->attr`.

**Returns:**

- EOK   Successful completion.
- EROFS  An attempt was made to utime 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**

<table>
<thead>
<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>

November 2, 2006
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

<table>
<thead>
<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_open()`,
`iofunc_write_verify()`, `resmgr_attach()`, `resmgr_context_t`,
`resmgr_io_funcs_t`

Writing a Resource Manager chapter of the *Programmer’s Guide*. 

November 2, 2006
iofunc_write_verify()  

Verify a client's write access to a resource

Synopsis:

```
#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.
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 *cip,
    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:

- **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\_MQQUEUE
  - \_\_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>
<th></th>
</tr>
</thead>
<tbody>
<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:**

- `iofunc_read_verify()`

  Writing a Resource Manager chapter of the *Programmer’s Guide.*
ionotify()

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:

```c
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.
- **ESRVRFAULT** 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

<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...
See also:

sigevent
IP

Synopsis:

```c
#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:

```c
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:

```c
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_RECVDSTADDR 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:

\[
\begin{align*}
\text{cmsg_len} & = \text{sizeof(struct cmsghdr)} + \text{sizeof(struct in_addr)} \\
\text{cmsg_level} & = \text{IPPROTO_IP} \\
\text{cmsg_type} & = \text{IP_RECVDSTADDR}
\end{align*}
\]

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:

\[
\begin{align*}
\text{cmsg_len} & = \text{sizeof(struct cmsghdr)} + \text{sizeof(struct sockaddr_dl)} \\
\text{cmsg_level} & = \text{IPPROTO_IP} \\
\text{cmsg_type} & = \text{IP_RECVIF}
\end{align*}
\]

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:

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:

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 `addr` is the local IP address of the desired interface or 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 `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:

```c
u_char loop; /* 0 = disable, 1 = enable (default) */
setsockopt(s, IPPROTO_IP, IP_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 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:

```c
struct ip_mreq mreq;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

where `mreq` is the following structure:

```c
class 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
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:

**ipsec.ah.cleartos**
When computing 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 AH behavior to interoperate with devices that implement RFC 1826 AH. Set this to a nonzero value (clear the type-of-service field) if you want to conform to RFC 2402.

**ipsec.ah.offsetmask**
When computing 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 AH behavior to interoperate with devices that implement RFC 1826 AH. Set this value to zero (clear the fragment offset field during computation) if you want to conform to RFC 2402.

**ipsec.dfbit**
Configures the socket manager behavior for IPv4 IPsec tunnel encapsulation. The variable is supplied to conform to RFC 2403 Chapter 6.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Changeable?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>net.inet.ipsec.ah.cleartos</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>net.inet.ipsec.ah.offsetmask</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>net.inet.ipsec.dfbit</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>net.inet.ipsec.ecn</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>net.inet.ipsec.debug</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>net.inet6.ipsec6.ecn</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td><code>net.inet6.ipsec6.debug</code></td>
<td>Integer</td>
<td>Yes</td>
</tr>
</tbody>
</table>
If the value is set to: Then:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The DF bit on the outer IPv4 header is cleared.</td>
</tr>
<tr>
<td>1</td>
<td>The outer DF bit on the header is set from the inner DF bit.</td>
</tr>
<tr>
<td>2</td>
<td>The DF bit is copied from the inner header to the outer.</td>
</tr>
</tbody>
</table>

`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:

\begin{verbatim}
  direction bypass
  direction entrust
  direction ipsec request ...
\end{verbatim}

where:

- **direction** The direction in which the policy is applied. It’s either \texttt{in} or \texttt{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 \texttt{setkey} (see the \textit{Utilities Reference}).
- **ipsec request ...**

Put the IPsec operation into the packet. You can specify one or more \texttt{request} strings using the following format:

\begin{verbatim}
  protocol(mode) src-dst[/level]
\end{verbatim}

For detailed descriptions of the arguments in the \texttt{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
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()`.
ipsec_dump_policy()

Returns:

A pointer to dynamically allocated string, or NULL if an error occurs.

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
ipsec_get_policylen()
Get the length of the IPsec policy

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>
<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:

IPsec, ipsec_dump_policy(), ipsec_set_policy(), ipsec_strerror()

setkey in the Utilities Reference
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.
方向 entrust
Consultation to SPD — defined by setkey.

方向 bypass
Bypass the IPsec processing, i.e. packets are transmitted in clear. This is for privileged sockets.

方向 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 sysct1().
  - **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: %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

<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:

IPsec, ipsec_dump_policy(), ipsec_get_policylen(), ipsec_strerror()

setkey in the Utilities Reference
Synopsis:
#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:
#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) {

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

<table>
<thead>
<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:

IPsec, ipsec_dump_policy(), ipsec_get_policylen(), ipsec_set_policy()
setkey in the Utilities Reference
Internet Protocol version 6

Synopsis:

```c
#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:

```c
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:

```c
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:

```c
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 cmsghdr structure containing this ancillary data, the cmsg_level argument is IPPROTO_IP6, 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_IP6, 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
options specified by a sticky option of the advanced API

options specified by using the basic API

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 IPv6 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 IPv6 sockets (i.e. for raw IPv6 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 IPv6 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.

```
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:**

```
#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:**

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

int main( void )
{
    if( isalnum( getchar() ) ) {
        printf( "That’s alpha-numeric!\n" );
    }

    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:**

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:

- `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;
}
```
**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()
**Synopsis:**

```c
#include <ctype.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 <ctype.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] );
        }
    }
}
```
```c
} else {
    printf( "Char %c is not an ASCII character\n", 
            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()
Test to see if a file descriptor is associated with a terminal

Synopsis:

```c
#include <unistd.h>

int isatty( int fildes );
```

Arguments:

- `fildes` 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 `fildes` is associated with a terminal.

Returns:

- `0` The `fildes` file descriptor doesn’t refer to a terminal.
- `1` The `fildes` 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 );
}
```
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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

open()
iscntrl()  © 2006, QNX Software Systems GmbH & Co. KG.

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:

`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] );
        }
    }
}
```
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

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(), 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] );
        }
    }
}
```
isdigit()

```c
} else {
    printf( "Char %c is not a digit character\n",
            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()`.

---

© 2006, QNX Software Systems GmbH & Co. KG.

November 2, 2006
Returns:

1  The filedes file descriptor matches fdtype.
0  The filedes 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()
isgraph()

Test a character to see if it's any printable character except a space

Synopsis:

```c
#include <cctype.h>

int isgraph(int c);
```

Arguments:

- `c` The character you want to test.

Library:

```c
libc
```

Use the `-l libc` 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 <cctype.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]);
        }
    }
}
```
```c
the_chars[i] );
} 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 not a printable character
Char ) is a printable 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(), isdigit(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
isinf(), isinff()

Test for infinity

Synopsis:

```c
#include <math.h>

int isinf ( double x );
int isinff ( 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 `isinff()` 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:

```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);
```


```c
d = 1.0/0.0;
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; `isinff()` is 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:**

`finite()`, `isnan()`
**islower()**

Test a character to see if it's a lowercase letter

**Synopsis:**

```c
#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:**

```c
#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];
}
}

return EXIT_SUCCESS;

}  

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

<table>
<thead>
<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()
**isnan(), isnanf()**

Test for not-a-number (NAN)

**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;
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
---
Cancellation point: No
Interrupt handler: No
Signal handler: No
Thread: Yes

See also:

finite(), isinf()
isprint()  © 2006, QNX Software Systems GmbH & Co. KG.

Test a character to see if it's any printable character, including a space

Synopsis:

    #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:

    #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
isprint()

the_chars[i] );
}
else {
    printf( "%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>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(), ispunct(), isspace(), isupper(), isxdigit(), tolower(), toupper()
ispunct()  © 2006, QNX Software Systems GmbH & Co. KG.
Test a character to see if it’s any punctuation character

Synopsis:

```c
#include <ctype.h>

int ispunct( int c );
```

Arguments:

- `c`  The character you want to test.

Library:

- `libc`

  Use the `-l libc` 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:

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

char 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] );
```

1476  C Library — I to L  November 2, 2006
ispunct()

```c
else {
    printf( "Char %c is not a punctuation character\n", 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

<table>
<thead>
<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(), 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>Yes</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…
See also:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(),
ispunct(), isupper(), isxdigit(), tolower(), toupper()
isupper()

Test a character to see if it's an uppercase letter

Synopsis:

```
#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:

```
#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()

```c

    the_chars[i] );
    }
    }

    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

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(), isxdigit(), tolower(), toupper()
iswalnum()  
Test a wide character to see if it’s alphanumeric

Synopsis:
```
#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>
</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 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.
iswalpha()  
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

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 \texttt{wchar\_t} arguments and \texttt{WEOF}.

See also:

\texttt{setlocale()}

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
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 `wctype()` to obtain `charclass`, or if a call to `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 `charclass` is 0.

**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 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.
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 `-l c` 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

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>

November 2, 2006 C Library — I to L 1491
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.
iswgraph()
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:**

```c
#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 \texttt{WEOF}.

See also:

\texttt{setlocale()} 

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**iswprint()**

Test a wide character to see if it’s any printable character, including space

**Synopsis:**

```c
#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 `wchar_t` arguments and WEOF.

See also:

`setlocale()`

“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:

```
#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>
</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 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.
**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**

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

*continued...*
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.
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

<table>
<thead>
<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.
iswxdigit()  
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>
</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 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()

Test a character to see if it's a hexadecimal digit

Synopsis:

```c
#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:

```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( isxdigit( the_chars[i] ) ) {
            printf( "Char %c is a hexadecimal digit",
```
```c
    the_chars[i] );
  } else {
    printf("Char %c is not a hexadecimal digit",
           the_chars[i] );
  }
}

return EXIT_SUCCESS;
}

produces the output:

Char A is a hexadecimal digit character
Char 5 is a hexadecimal digit character
Char $ is not a hexadecimal digit character

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:

isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), tolower(), toupper()
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:

```c
#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>
<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:

* atoi(), atol(), ltoa(), sscanf(), strtol(), strtoul(), ultoa(), utoa()
Compute a Bessel function of the first kind

Synopsis:
```
#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:
```
#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{j0()} is POSIX 1003.1 XSI; \textit{j0f()} is Unix

\begin{tabular}{|l|c|}
\hline
Safety & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\hline
\end{tabular}

See also:

\textit{errno, j1(), jn(), y0(), y1(), yn()}

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

<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(), jn(), y0(), y1(), yn()
**jn(), jnf()**

Compute a Bessel function of the first kind

**Synopsis:**
```
#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:**
```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y, z;
```
\begin{verbatim}
x = j0(2.4);
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

<table>
<thead>
<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{errno, j0(), j1(), y0(), y1(), yn()}
Generate a pseudo-random signed long integer in a thread-safe manner

**Synopsis:**
```
#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>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(), 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:

<table>
<thead>
<tr>
<th>If <strong>pid</strong> is:</th>
<th>Then <strong>sig</strong> is sent to:</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 -pid</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 pid causes sig to be generated for the sending process, and if sig isn’t blocked, either sig or at least one pending unblocked signal is delivered before the 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 (errno is set).

**Errors:**

- **EAGAIN** Insufficient system resources are available to deliver the signal.
- **EINVAL** The sig is invalid.
- **EPERM** The process doesn’t have permission to send this signal to any receiving process.
- **ESRCH** The given pid doesn’t exist.

**Examples:**

See *sigprocmask()*.

**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:

`getpid()`, `killpg()`, `setsid()`, `sigaction()`, `signal()`, `SignalKill()`, `sigqueue()`
#killpg()

**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:

```c
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 \texttt{sig} is invalid or not supported.

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 \texttt{pgrp} or \texttt{pgrp} is less than or equal to 1.

\textbf{Examples:}

See \texttt{sigprocmask()}.  

\textbf{Classification:}

POSIX 1003.1 XSI

\begin{tabular}{|l|l|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}

\textbf{See also:}

\texttt{getpid()}, \texttt{kill()}, \texttt{setsid()}, \texttt{sigaction()}, \texttt{signal()}, \texttt{SignalKill()}, \texttt{sigqueue()}


Calculate the absolute value of a long integer

Synopsis:

```c
#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:

```c
#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
```
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:

abs(), cabs(), fabs()
Synopsis:

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

int lchown( 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 `lchown()` 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, `lchown()` 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_atime 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

Cancellation point  No
Interrupt handler   No
Signal handler      Yes
Thread              Yes

See also:

chmod(), chown(), errno, fchown(), fstat(), open(), stat()
Initialize a sequence of pseudo-random numbers

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

<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>
</tbody>
</table>

continued…
Icong48()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</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:

drand48(), erand48(), jrand48(), lrand48(), mrand48(), nrand48(), seed48(), srand48()
**ldexp(), ldexpf()**  

Multiply a floating-point number by an integral power of 2

**Synopsis:**
```
#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 `-l m` 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

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

See also:

- frexp()
- modf()
ldiv() — Perform division on long integers

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:

```c
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;

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

<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:

div()
Synopsis:

```
#include <search.h>

void * lfind( const void *,
        const void *,
        unsigned *,
        unsigned,
        int (*) (const void *,
                 const void *) ) ;
```

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:

**libc**

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

The `lfind()` function returns a pointer into a table indicating where an entry may be found.

The `lfind()` function is the same as `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 `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' is not a C keyword\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

<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:

bsearch(), lsearch()
Synopsis:

```c
#include <math.h>

double lgamma( double x );

double lgamma_r( double x, int* signgam );

float lgammaf( float x );

float lgammaf_r( float x, int* signgam );
```

Arguments:

- `x` An arbitrary number.
- `signgam` (lgamma_r(), lgammaf_r() only) A pointer to a location where the function can store the sign of Γ(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 Γ function and are equivalent to `gamma()`. These functions return \( \ln |\Gamma(x)| \), where \( \Gamma(x) \) is defined as follows:

\[
\int_0^\infty e^{-t} t^{x-1} dt
\]

For \( x > 0 \):

For \( x < 1 \):

\[
n / (\Gamma(1-x) * \sin(nx))
\]
The results converge when $x$ is between 0 and 1. The $\Gamma$ function has the property:

$$\Gamma(N) = \Gamma(N-1) \times N$$

The $lgamma^*$ 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:

```c
    g = signgam * exp( lgamma( x ));
```

to compute $g = \Gamma(x)'$. Instead, compute $lgamma()$ first:

```c
    lg = lgamma(x);
    g = signgam * 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 $errno$ 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;

erro = EOK;
a = 0.5;
b = lgamma(a);
printf("lgamma(%f) = %f %d \n", a, b, erro);
    return(0);

}
```

produces the output:

```
lgamma(0.500000) = 0.572365 0
```

Classification:

`lgamma()` and `lgammaf()` are ANSI, POSIX 1003.1; `lgamma_r()` and `lgammaf_r()` are 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:

gamma()
Synopsis:

```c
#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 argv[2] doesn't already exist.
 */
#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

<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`, `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;
lio_listio() — 1550

_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_.

© 2006, QNX Software Systems GmbH & Co. KG.
Classification:

POSIX 1003.1 AIO

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:

aio_cancel(), aio_error(), aio_fsync(), aio_read(), aio_return(),
aio_write(), close(), exec(), execle(), execle(), execle(), execv(),
execve(), execvp(), execvpe(), exit(), fcntl(), fork(), lseek(), read(),
sigevent, write()
**listen()**

*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:**

0  Success.

-1  An error occurred (`errno` is set).

**Errors:**

EBADF  Invalid descriptor `s`.

EOPNOTSUPP  The socket isn’t of a type that supports the `listen()` operation.

**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:**

`accept()`, `connect()`, `socket()`
localeconv()  © 2006, 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.

cchar * mon_thousands_sep
   The character used to separate groups of digits on the left of the
decimal-point character in formatted monetary quantities.

cchar * mon_grouping
   A string whose elements indicate the size of each group of
digits in formatted monetary quantities.

cchar * grouping
   A string whose elements indicate the size of each group of
digits in formatted nonmonetary quantities.

cchar * positive_sign
   The string used to indicate a nonnegative monetary quantity.

cchar * negative_sign
   The string used to indicate a negative monetary quantity.

cchar int_frac_digits
   The number of fractional digits (to the right of the decimal
digit point) to display in an internationally formatted monetary
quantity.

cchar frac_digits
   The number of fractional digits (to the right of the decimal
digit point) to display in a formatted monetary quantity.

cchar p_cs_precedes
   Set to 1 or 0 if the currency_symbol precedes or follows the
value for a nonnegative monetary quantity.

cchar 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.
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()  © 2006, QNX Software Systems GmbH & Co. KG.

```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()`, `strftime()`, `strcmp()`, `strcoll()`, `strcpy()`, `strchr()`, `strcspn()`, `strtok()`
Synopsis:

```
#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
Returns:

A pointer to the static `tm` structure containing the time information.

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:

`asctime()`, `asctime_r()`, `clock()`, `ctime()`, `ctime_r()`, `difftime()`, `gmtime()`, `gmtime_r()`, `mktime()`, `localtime()`, `strftime()`, `time()`, `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.
**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
Synopsis:

```c
#include <unistd.h>

int lockf( int filedes,
           int function,
           off_t size );
```

Arguments:

- **filedes**
  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:

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

Description:

You can use the \texttt{lockf()} function to lock a section of a file, using advisory-mode locks. If other threads call \texttt{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 \texttt{lockf()} is supported for regular files and may be supported for other files.

The sections locked with \texttt{F\_LOCK} or \texttt{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.

\texttt{F\_ULONG} 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 \texttt{size} bytes or to the end of file if \texttt{size} is \texttt{(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.
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_UNLOCK request of size 0. Otherwise an F_UNLOCK 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()  © 2006, QNX Software Systems GmbH & Co. KG.

EINVAL  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.

ENOMEM  The system can’t allocate sufficient memory to store lock resources.

EOPNOTSUPP or EINVAL  
The implementation doesn’t support the locking of files of the type indicated by fildes.

EOVERFLOW  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

<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(), flock(), open()
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>
</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(), log10(), pow()`
Synopsis:

```c
#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><code>x = NAN</code></td>
<td><code>NAN</code></td>
</tr>
<tr>
<td><code>x &lt; -1.0</code></td>
<td><code>-HUGE_VAL</code>, or <code>NAN</code> (errno is set to <code>EDOM</code>).</td>
</tr>
<tr>
<td><code>x = -1.0</code></td>
<td><code>-HUGE_VAL</code> (errno may be set to <code>ERANGE</code>).</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 \textit{errno} to 0, call the function, and then check \textit{errno} again. These functions don’t change \textit{errno} if no errors occurred.

\section*{Classification:}
ANSI, POSIX 1003.1

\section*{Safety}
\begin{tabular}{|l|l|}
\hline
\textbf{Cancellation point} & \textit{No} \\
\hline
\textbf{Interrupt handler} & \textit{No} \\
\hline
\textbf{Signal handler} & \textit{No} \\
\hline
\textbf{Thread} & \textit{Yes} \\
\hline
\end{tabular}

\section*{See also:}
\textit{ilogb()}, \textit{log()}, \textit{logb()}, \textit{log10()}

\textcopyright{} 2006, QNX Software Systems GmbH & Co. KG.
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

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`, `exp()`, `log()`, `pow()`


Synopsis:

```c
#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:

```c
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 <code>x</code> is:</th>
<th><code>logb()</code> returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>-HUGE_VAL (\texttt{errno} is set to EDOM)</td>
</tr>
<tr>
<td>&lt;0.0</td>
<td>-HUGE_VAL (\texttt{errno} may be set to ERANGE)</td>
</tr>
</tbody>
</table>

continued...
logb(), logbf()

If \( x \) is: \n\[
\begin{array}{l}
\pm \infty & \text{returns:} & +\infty \\
\end{array}
\]

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.

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

Cancellation point  No

continued...
logb(), logbf()

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:

ilogb(), log(), log10(), log1p()
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.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

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.
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:

forkpty(), ioctl(), openpty()
**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 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 following output:

```
after setjmp 0
about to longjmp
back from longjmp 14
```
### longjmp()

---

**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:**

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:

```
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 **p1 = (const char **) op1;
    const char **p2 = (const char **) op2;
    return( strcmp( *p1, *p2 ) );
}
```
Using the program above, this input:

one two one three four

produces the output:

one
two
three
four

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:

bsearch(), lfind()
lseek(), lseek64()

Set the current file position at the operating-system level

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
Iseek(), Iseek64()

Safety

<table>
<thead>
<tr>
<th>Component</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(), execle(), execlp(), execlepe(), execv(), execvp(), execve(), execvpe(), 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:

/*
 * 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

<table>
<thead>
<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:

*errno*, *fstat()*(*), *readlink()*(*), *stat()*
Synopsis:

```
#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:

- **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 115033
8 30735
10 12765
12 7479
14 491b
16 31dd
```
Classification:

\texttt{ltoa()} is QNX 4; \texttt{lltoa()} is Unix

\begin{center}
\begin{tabular}{l|c}
\hline
Safety & \\
\hline
Cancellation point & No \\
Interrupt handler & Yes \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}
\end{center}

See also:

\texttt{atoi()}, \texttt{atol()}, \texttt{iota()}, \texttt{sscanf()}, \texttt{strtol()}, \texttt{strtoul()}, \texttt{ultoa()}, \texttt{utoa()}
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 \textit{fd}

Examples:

\begin{verbatim}
#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;
}
\end{verbatim}
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>

Caveats:

The `ltrunc()` function isn’t portable, and shouldn’t be used in new code. Use `ftruncate()` instead.

See also:

`errno`, `ftruncate()`, `lseek()`
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 ltruncf()</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>
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**

<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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

abort(), atexit(), argc, argv, auxv, close(), exec(), execle(), execlpe(), execv(), execve(), execvp(), execvpe(), exit(), exit(), getenv(), putenv(), sigaction(), signal(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnv(), spawnve(), spawnvp(), spawnvpe(), system(), wait(), waitpid()
Synopsis:

```c
#include <malloc.h>

struct mallinfo mallinfo ( void );
```

Library:

```c
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 hblkhed;  /* space in header block headers */
    int usmblks;  /* space in small blocks in use */
    int fsmblks;  /* memory in free small blocks */
    int uordblks; /* space in big blocks in use */
    int fordblks; /* memory in free big blocks */
    int keepcost; /* penalty if M_KEEP is used */
    /* not used */
};
```

Returns:

A `struct mallinfo`.

Classification:

QNX Neutrino
mallinfo()

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(), 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 `_amblksiz` 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.
malloc()

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

<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:

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()
malloc()

The Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino *Programmer’s Guide.*
mallopt()  © 2006, QNX Software Systems GmbH & Co. KG.
Control the extra checking for memory allocation

Synopsis:

```c
#include <malloc.h>

int mallopt( int cmd,
            int value );
```

Arguments:

- **cmd** Options used to enable additional checks in the library.
  - MALLOC_CKACCESS
  - MALLOC_FILLAREA
  - MALLOC_CKCHAIN
  - MALLOC_VERIFY
  - MALLOC_FATAL
  - MALLOC_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:

- **MALLOC_CKACCESS**
  Turn on (or off) boundary checking for memory and string operations.
Environment variable: **MALLOC_CKACCESS**.
The value argument can be:
- zero to disable the checking
- nonzero to enable it.

**MALLOC_FILLAREA**
Turn on (or off) fill-area boundary checking.
Environment variable: **MALLOC_FILLAREA**.
The value argument can be:
- zero to disable the checking
- nonzero to enable it.

**MALLOC_CKCHAIN**
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: **MALLOC_CKCHAIN**.
The value argument can be:
- zero to disable the checking
- nonzero to enable it.

**MALLOC_VERIFY**
Perform a chain check. If an error is found, perform error handling. The value argument is currently ignored; pass 1 for it.

**MALLOC_FATAL**
Specify the **malloc** fatal handler.
Environment variable: **MALLOC_FATAL**.
Use one of the following for the value arguments:
Symbolic name | Value | Description
---|---|---
M_HANDLE_IGNORE | 0 | Cause the program to dump a core file.
M_HANDLE_ABORT | 1 | Terminate execution with a call to `abort()`
M_HANDLE_ABORT | 2 | Exit immediately
M_HANDLE_CORE | 3 | Cause the program to dump a core file
M_HANDLE_SIGNAL | 4 | Stop the program when this error occurs

**MALLOC_WARN**

Specify the `malloc` warning handler. The values are similar to `MALLOC_FATAL`; see above.

Environment variable: `MALLOC_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.

**Returns:**

0 on success, or -1 if an error occurs (`errno` is set).

**Classification:**

QNX Neutrino
mallopt()

Safety

Cancellation point  No
Interrupt handler  No
Signal handler  No
Thread  Yes

See also:

calloc(), free(), mallinfo(), malloc(), realloc()

Heap Analysis: Making Memory Errors a Thing of the Past chapter of the Neutrino Programmer's Guide
Return the greater of two numbers

Synopsis:

```
#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.

Note:
The `max()` function is for C programs only. For C++ programs, use the `_max()` or `_min()` macros.

Examples:

```
#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

continued...
**max()**

<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:**

`min()`
**mblen()**

Count the bytes in a multibyte character

**Synopsis:**

```c
#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.
mblen()

> 0  The number of bytes that comprise the multibyte character (if the next \( n \) or fewer bytes form a valid multibyte character).

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

Cancellation point  No

continued...
mblen() © 2006, 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>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:

```
#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.
EINV AL The ps argument points to an invalid object.

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

“Character manipulation functions” and “Wide-character functions” in the summary of functions chapter.
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.
mbtowc() © 2006, QNX Software Systems GmbH & Co. KG.

Returns:

\[(\text{size}\_\text{t})-2\] After converting all \(n\) characters, the resulting conversion state indicates an incomplete multibyte character.

\[(\text{size}\_\text{t})-1\] The function detected an encoding error before completing the next multibyte character; the function sets \textit{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.

EINVAL The \(ps\) argument points to an invalid object.

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>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.
**Synopsis:**

```
#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 libc` 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()`
- `wcstombs()`
- `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 \(^*ps\) describes an initial conversion state; otherwise zero.

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:

\[mbrlen(), mbtowc(), mbsrtowcs(), mbstowcs(), wcsrtoi()\]

“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
Or:

- Converting the next character would exceed the limit of \( n \) total bytes.

This function is affected by LC_TYPE.

**Returns:**

\[(\text{size}_t)-1\] Failure; invalid wide-character code.

\( x \) Success; the number of total bytes successfully converted, not including the terminating NULL byte.

**Errors:**

EILSEQ Invalid character sequence.

EINV AL 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\n", wc, len );
        for( i = 0; i < len; i++ ) {
            printf( "/%4.4x", wbuffer[i] );
        }
        printf( "\n" );
    }

    return EXIT_SUCCESS;
}
```

This produces the output:

```
string(6)
/0073/0074/0072/0069/006e/0067
```

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No

continued...
mbstowcs()

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:

erro

“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 `mbrtowc()` 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.
mbtowc()

≠ 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( "\t%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

**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`

“Multibyte character functions” and “Wide-character functions” in the summary of functions chapter
enable memory allocation routine consistency checks

Synopsis:

```c
#include <malloc.h>

int mcheck(
    void (*)(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

**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:

`mallopt()`, `mprobe()`

Heap Analysis in the *Programmer’s Guide*
mem_offset(),
mem_offset64()

Get the offset of a mapped typed memory block

Synopsis:

```
#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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addr</code></td>
<td>The address of the memory block whose offset and contiguous length you want to get.</td>
</tr>
<tr>
<td><code>fd</code></td>
<td>The file descriptor that identifies the typed memory object. This must be the descriptor that you used (in a call to <code>mmap()</code>) to establish the mapping that contains <code>addr</code>.</td>
</tr>
<tr>
<td><code>length</code></td>
<td>The length of the block of memory that you want the offset for.</td>
</tr>
<tr>
<td><code>offset</code></td>
<td>A pointer to a location where the function can store the offset of the memory block.</td>
</tr>
<tr>
<td><code>contig_len</code></td>
<td>A pointer to a location where the function can store either <code>length</code> or the length of the largest contiguous block of typed memory that’s currently mapped to the calling process starting at <code>addr</code>, whichever is smaller.</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 `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()`.

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.

If the physical address is not a valid offset 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.

Returns:

0 Success.

-1 An error occurred (`errno` is set).
mem_offset(), mem_offset64()

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:

```c
#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;
}
```

Classification:

mem_offset() is QNX Neutrino; mem_offset64() is Large-file support

Safety

Cancellation point    No
Interrupt handler    No
continued…
**mem_offset()**, **mem_offset64()**

<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:**

`mmap()`, `posix_mem_offset()`, `posix_mem_offset64()`
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()
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:

```bash
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

<table>
<thead>
<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`, `memcmp`, `memcpy`, `memicmp`, `memmove`, `memset`
memchr()

Find the first occurrence of a character in a buffer

Synopsis:

```c
#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:

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

int main( void )
{
    char buffer[80];
    char* where;
```
```c
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**

<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()`, `memcmp()`, `memcpy()`, `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

- Cancellation point: No
- Interrupt handler: Yes
- Signal handler: Yes
- Thread: Yes
See also:

`memccpy()`, `memchr()`, `memcpy()`, `memcmp()`, `memmove()`, `memset()`
memcpy()

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 )
{
    char buffer[80];

    memcpy( buffer, "Hello", 5 );
    buffer[5] = '\0';
    printf( "%s\n", 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>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(), memicmp(), memmove(), memset()
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`. 
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

<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()`, `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;

    strcpy( buffer, "World" );

    retval = memicmp( 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

<table>
<thead>
<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(), memchr(), memcmp(), memcpy(), memmove(), memset()
**Synopsis:**

```c
#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:**

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

int main( void )
{
    char buffer[80];
```
```
strcpy(buffer, "World");
memmove(buffer+1, buffer, 79);
printf("%s\n", buffer);

return EXIT_SUCCESS;
}
```

produces the output:

```
WWorld
```

**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:**

`memccpy()`, `memchr()`, `memcpy()`, `memcmp()`, `memset()`, `wmemmove()`
**memset()**

Set memory to a given value

**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 `-l c` 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 );
}
```


```c
    memset(buffer, 79, '\0');
    puts(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>

See also:

`memccpy()`, `memchr()`, `memcmp()`, `memcpy()`, `memcmp()`, `memmove()`
message_attach()

Attach a message range

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.
**message_attach()**

**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:

```c
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.
message_attach()

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 *tp;
    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",}
message_attach()
/* Initialize an event structure, and attach a pulse to it */
if((event.sigev_code = pulse_attach(dpp,
    MSG_FLAG_ALLOC_PULSE, 0,
    &timer.tick, NULL)) == -1) {
    fprintf(stderr, "Unable to attach timer pulse.\n");
    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
message_attach()

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:

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
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_SIDE_CHANNEL — 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()`.

© 2006, QNX Software Systems GmbH & Co. KG.
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;
    }
```

☞ 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;
    }
```
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
message_detach()

Detach a message range

Synopsis:

```c
#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:

EINVVAL 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.
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

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>

**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.

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`:

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    (void)mkdir( "/hd/src",
                S_IRWXU |
                S_IRGRP  | S_IXGRP |
                S_IROTH  | S_IXOTH );

    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:

`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_atime`, `st_mtime`, `st_atime` and `st_mtime` fields of the file are marked for update. Also, the `st_cctime` and `st_jmtime` fields of the directory that contains the new entry are marked for update.
mkfifo()

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.
EROF S  The named file resides on a read-only filesystem.

Examples:

```c
#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()

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:

```c
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

<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:

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

<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:

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()*.  

1692  C Library — M to O  November 2, 2006
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

<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(), 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:
<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 \texttt{RLIMIT\_MEMLOCK rlimit}.

- \texttt{EAGAIN}: Some or all of the memory identified by the operation couldn’t be locked when the call was made.
- \texttt{EPERM}: The calling process doesn’t have the superuser capabilities.

**Classification:**

- **POSIX 1003.1 MLR**

**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{mlockall()}, \texttt{mmap()}, \texttt{munlock()}
**Synopsis:**

```
#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.
**mlockall()**

*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:

- Call `mlockall(MCL_FUTURE)`, followed by one or more calls to `mmap()`
• Optionally call `mmap()`, and then call
  \texttt{mlockall(MCL\_CURRENT)}

\textbf{Returns:}

\begin{itemize}
  \item \texttt{0} \hspace{1em} Success.
  \item \texttt{-1} \hspace{1em} An error occurred (\texttt{errno} is set).
\end{itemize}

\textbf{Errors:}

\begin{itemize}
  \item \texttt{EAGAIN} \hspace{1em} Some or all of the memory identified by the operation couldn’t be locked when the call was made.
  \item \texttt{EINVAL} \hspace{1em} The \texttt{flags} argument is zero.
  \item \texttt{ENOMEM} \hspace{1em} 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 \texttt{RLIMIT\_MEMLOCK rlimit}.
  \item \texttt{EPERM} \hspace{1em} The calling process doesn’t have the appropriate privilege to perform the requested operation.
\end{itemize}

\textbf{Classification:}

POSIX 1003.1 ML

\textbf{Safety}

\begin{tabular}{ll}
  \textbf{Cancellation point} & No \\
  \textbf{Interrupt handler} & No \\
  \textbf{Signal handler} & Yes \\
  \textbf{Thread} & Yes \\
\end{tabular}
mlockall()

See also:

mlock(), mmap(), munlock()
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()

fidles  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.
mmap(), mmap64()

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.

MAP_PHYS  Physical memory is required. The fildes parameter must be NOFD. When used with MAP_PRIVATE or

November 2, 2006  C Library — M to O  1705
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:

```c
/* Map in VGA display memory */
addr = mmap( 0,
            65536,
            PROT_READ|PROT_WRITE,
            MAP_PHYS|MAP_SHARED,
            NOFD,
            0x0a0000 );
```

To allocate a DMA buffer for a bus-mastering PCI network card:

```c
/* 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).

- **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
",
                strerror(errno));
        return EXIT_FAILURE;
    }
    printf("Map addr is %p
", (void*) addr);

    for (i = 0; i < 3 * 80; ++i) {
        c = *addr++;
        if (c >= ' ' && c <= 0x7f)
            putchar(c);
        else
            putchar('.');
    }
    return EXIT_SUCCESS;
}
```

Classification:

`mmap()` is POSIX 1003.1 MF SHM TYM; `mmap64()` is Large-file support
mmap(), mmap64()

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:

mmap_device_io(), mmap_device_memory(), munmap()
mmap_device_io()

Gain access to a device’s registers

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.
**mmap_device_io()**

**Classification:**

QNX Neutrino

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**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() © 2006, QNX Software Systems GmbH & Co. KG.
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:

☞ 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.

- PROT_NONE — the region can’t be accessed.
- PROT_READ — the region can be read.
- PROT_WRITE — the region can be written.
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:

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:**

```c
/* map in the physical memory, 0xb8000 is text mode VGA video memory */

ptr = mmap_device_memory( 0, len, PROT_READ|PROT_WRITE|PROT_NOCACHE, 0, 0xb8000 );
if ( ptr == MAP_FAILED ) {
    perror( "mmap_device_memory for physical address 0xb8000 failed" );
    exit( EXIT_FAILURE );
}
```

**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:

mmap(), mmap_device_io(), munmap_device_memory()
modem_open()  © 2006, QNX Software Systems GmbH & Co. KG.
Open a serial port

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|LIFLOW|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.
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.
ENOSYS  The `modem_open()` function isn't implemented for the filesystem specified in `device`.

ENOTDIR  A component of the path prefix isn't a directory.

ENXIO  No process has the file open for reading.

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*", "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, NULL},
{0, 0, 0, 0, 0, 0, 1, NULL},
{0, 0, 0, 0, 0, 2, NULL},
{0, 0, 0, 0, 0, 3, NULL},
{0, 0, 0, 0, 0, 4, NULL},
{ NULL }
};

void io(char* progress, char* in, char* out) {
    if(progress)

```
```c

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>
</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>
modem_open()

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 followed by more characters. Without this flag, 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.

**cancel**

NULL, or a callback that’s called whenever the quiet time period expires while waiting for more input.

**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 `modem_read()` function reads up to bufsize bytes from the device specified by the file descriptor, `fd`, and places them into the buffer pointed to by `buf`.

If no characters are received within the given timeout, `modem_read()` returns with -1.

When at least one character has been received, `modem_read()` returns if the flow of incoming characters stops for at least the quiet time period. The number of characters saved in `buf` is returned.

If you provide a `cancel` function, it’s called once each quiet time period while waiting for input. If this function returns a nonzero value, `modem_read()` returns -1 immediately and sets `errno` to `ETIMEDOUT`. You can use the `cancel` function as a callback in a graphical dialer that needs to support a cancel button to stop a script (see `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.
- **EINTR** 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

<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:**

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()
Synopsis:

```c
#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 `libc.a`, but not in `libc.so` (in order to save space).

**Description:**

The `modem_script()` function runs the script `table` on the device associated with the file descriptor `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 `io()` as follows:

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*io)(str, 0, 0)</td>
<td>Emitted progress string</td>
</tr>
<tr>
<td>(*io)(0, str, 0)</td>
<td>Received string</td>
</tr>
<tr>
<td>(*io)(0, 0, str)</td>
<td>Emitted 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 `cancel` function, it’s called once each `newquiet` 1/10 of a second while waiting for input. If this function returns a nonzero value, the read returns immediately with -1 and `errno` is set to ETIMEDOUT. You can use the `cancel` function as a callback in a graphical dialer that needs to support a cancel button to stop a script.

The `table` is an array of `modem_script` structures that contain the following members:

- **char curstate**
  
  The current state. Execution always begins at state 1, which must be the first array element of `table`. Multiple elements may have the same current 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`. 

---

**November 2, 2006**

---

© 2006, QNX Software Systems GmbH & Co. KG.
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 retval 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, NULL, "*$ *"},
{0, 0, 0, 0, 0, 0, 1, NULL, "*no carrier*"},
{0, 0, 0, 0, 0, 2, NULL, "*no answer*"},
{0, 0, 0, 0, 0, 3, NULL, "*no dialtone*"},
{0, 0, 0, 0, 4, NULL, "*busy*"},
{ 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 <strong>ATDT1-591-0934</strong>. 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>retvalue</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>retvalue</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>retvalue</code> (3).</td>
</tr>
<tr>
<td><em>busy</em></td>
<td>Go to state 0 (the termination newstate), return with the contents of <code>retvalue</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 <code>MODEM_LASTLINE</code>, 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 <code>MODEM_BAUD</code>, 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>
<tr>
<td><em>no dialtone</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>
<tr>
<td><em>no dialtone</em></td>
<td>Same as previous table</td>
</tr>
</tbody>
</table>

*continued…*
Input | Action
---|---
*busy* | Same as previous table

State 5

| Input | Action |
---|---|
*$* | Go to state 0 (the termination newstate), return with the contents of retvalue (0).|
*no carrier* | Same as previous table|
*no answer* | Same as previous table|
*no dialtone* | Same as previous table|
*busy* | Same as previous 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 retvalue 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:

| Error Code | Description |
---|---|
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:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the write operation.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The file descriptor, fildes, isn’t a valid file descriptor open for writing.</td>
</tr>
<tr>
<td>EINTR</td>
<td>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.</td>
</tr>
<tr>
<td>EIO</td>
<td>A physical I/O error occurred. The precise meaning depends on the device.</td>
</tr>
<tr>
<td>EPIPE</td>
<td>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.</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:

`modem_open()`, `modem_read()`, `modem_script()`
modf(), modff()

Break a number into integral and fractional parts

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:**

```c
#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**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
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_NO_CREAT — 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() — call resmgr_attach() with RESMGR_FLAG_AFTER.

- _MOUNT_OPAQUE — 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.
Returns:

-1 on failure; no server supports the request (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>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() © 2006, QNX Software Systems GmbH & Co. KG.
Strip off common mount arguments

Synopsis:

```
#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…
### mount_parse_generic_args()

<table>
<thead>
<tr>
<th>Option</th>
<th>Set/Clear 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 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 (default).</td>
</tr>
<tr>
<td>suid</td>
<td>Clear _MOUNT_SUID</td>
<td>Honor setuid bits on the filesystem (default).</td>
</tr>
<tr>
<td>update</td>
<td>Set _MOUNT_REMOUNT</td>
<td>This path is already mounted, perform an update.</td>
</tr>
</tbody>
</table>
**mount_parse_generic_args()** © 2006, QNX Software Systems GmbH & Co. KG.

**Returns:**

A string pointing to unprocessed options.

**Examples:**

```c
while ((c = getopt(argc, argv, "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

<table>
<thead>
<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:**

`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.
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:

\textit{mallopt()}, \textit{mcheck()}

The Heap Analysis chapter in the \textit{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.
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:

```
#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 `-l mq` 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

<table>
<thead>
<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_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 \texttt{mq} and \texttt{mqueue} in the \textit{Utilities Reference}.

We recommend that you use the following event types in this case:

- \texttt{SIGEV\_SIGNAL}
- \texttt{SIGEV\_SIGNAL\_CODE}
- \texttt{SIGEV\_SIGNAL\_THREAD}
- \texttt{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 \texttt{errno} is set to \texttt{EBUSY}.

If a process has registered for notification, and another process is blocked on \texttt{mq\_receive()}, then the \texttt{mq\_receive()} call is satisfied by any arriving message. The resulting behavior is as if the message queue remained empty.

If \texttt{notification} is NULL and the current process is currently registered for notification, then the existing registration is removed.

\textbf{Returns:}

-1 if an error occurred (\texttt{errno} is set). Any other value indicates success.

\textbf{Errors:}

- \texttt{EBADF} Invalid message queue \texttt{mqdes}.
- \texttt{EBUSY} A process has already registered for notification for the given queue.
mq_notify()

Classification:

POSIX 1003.1 MSG

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:

mq_open(), mq_receive(), mq_send(), sigevent

mq, mqueue in the Utilities Reference
mq_open()
Open a message queue

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.

If `mq_attr` is `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><code>mq_maxmsg</code></td>
<td>1024</td>
<td>64</td>
</tr>
<tr>
<td><code>mq_msgsize</code></td>
<td>4096</td>
<td>256</td>
</tr>
<tr>
<td><code>mq_flags</code></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 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_open()` function opens a message queue referred to by `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 `mq` and `mqueue` in the 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><code>cwd/entry</code></td>
</tr>
<tr>
<td>/entry</td>
<td><code>/dev/mqueue/entry</code> using the traditional (<code>mqueue</code>) implementation, or <code>/dev/mq/entry</code> using the alternate (<code>mq</code>) implementation</td>
</tr>
<tr>
<td>entry/newentry</td>
<td><code>cwd/entry/newentry</code></td>
</tr>
<tr>
<td>/entry/newentry</td>
<td><code>/entry/newentry</code></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 queue on another node, you have to use the traditional (`mqueue`) implementation and specify the name as `/net/node/mqueue_location`.

If `name` doesn’t exist, `mq_open()` examines the third and fourth parameters: a `mode_t` and a pointer to an `mq_attr` structure.

The only time that a call to `mq_open()` with `O_CREAT` 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 *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.
- **EINTR** The operation was interrupted by a signal.
- **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

<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_getattr(), mq_notify(), mq_receive(), mq_send(),
mq_setattr(), mq_timedreceive(), mq_timedsend(), mq_unlink()

mq, mqueue in the Utilities Reference
**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 mq and mqueue in the Utilities Reference.

If you call mq_receive() with a msg_len of anything other than the mq_msgsize of the specified queue, then mq_receive() returns an error, and errno is set to EINVAL.

If there are no messages on the queue specified, and O_NONBLOCK wasn’t set in oflag during mq_open(), then the mq_receive() call blocks. If multiple mq_receive() 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_receive() with a NULL 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 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 for the traditional
**mq_receive()**

(mqueue) implementation, and 256 bytes for the alternate (mq) implementation.

**EMSGSIZE** The given msg_len is shorter than themq_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

<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:**

mq_close(), mq_open(), mq_send(), mq_timedreceive(), read()

mq, mqueue in the Utilities Reference
Synopsis:

```c
#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

<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:**

mq_close(), mq_open(), mq_receive(), mq_timedsend()

mq, mqueue in the Utilities Reference
**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>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>

### See also:

- `mq_getattr()`, `mq_open()`, `mq_receive()`, `mq_send()`
- `mq` and `mqueue` in the *Utilities Reference*
**Synopsis:**

```c
#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.
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.
**mq_timedreceive()**

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.
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:

- **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` isn’t opened for writing.
- **EINTR**: The call was interrupted by a signal.
mq_timedsend()

EINVAL One of the following is true:

- The msg_len is negative.
- The msg_prio is greater than (MQ_PRIO_MAX - 1).
- The msg_prio is less than 0.
- The MQ_PRIO_RESTRCT flag is set in the
  mq_attr member of mqdes, and msg_prio is
  greater than the priority of the calling process.

EMSGSIZE The msg_len argument is greater than the msgsize
  associated with the specified queue.

ETIMEDOUT The timeout value was exceeded.

Examples:
See the example for mq_timedreceive().

Classification:
POSIX 1003.1 MSG

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:

mq_close(), mq_open(), mq_receive(), mq_send(), mq_timedreceive(),
timespec

mq.mqueue in the Utilities Reference
mq_unlink()

Remove a queue

Synopsis:

```c
#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

<table>
<thead>
<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_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 {
    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.

**To:**  Use this function:

| Compare two `nd` objects | `ND_NODE_CMP()` |

continued…
### To:

| Convert a *nd* to text | `netmgr_ndtostr()` |
| Convert text to a *nd* | `netmgr_strtond()` |

### Classification:

QNX Neutrino

### See also:

`MsgInfo()`, `MsgReceive()`, `MsgReceivev()`, `ND_NODE_CMP()`, `netmgr_ndtostr()`, `netmgr_remote_nd()`, `netmgr_strtond()`
**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 B
"B" blocked on "C", "C" blocked on "B"
DEADLOCK
C sends to A
"A" blocked on "B", "B" blocked on "C", "C" blocked on "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>
<tr>
<td>B sends pulse to → A</td>
<td>Requested data available</td>
</tr>
<tr>
<td>A sends to → B</td>
<td>Request for the data</td>
</tr>
<tr>
<td>B replies to → A</td>
<td>Reply with data</td>
</tr>
</tbody>
</table>

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-qnet`s.

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_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:
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"

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:

/* 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;

---

**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:**

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(), сигеvent, SignalWaitinfo())
_msgError(), _msgError_r()

Unblock a client and set its errno

Synopsis:

```
#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().
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 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

<table>
<thead>
<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>

November 2, 2006
See also:

`ChannelCreate()`, `MsgRead()`, `MsgReadv()`, `MsgReceive()`,
`MsgReceivev()`, `MsgSend()`, `MsgSendv()`
**Synopsis:**

```
#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*()`.

© 2006, QNX Software Systems GmbH & Co. KG.
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 `revid` 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()`, `msg_info`, `MsgRead()`, `MsgReady()`,
`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`.  

**Library:**

`libc`

Use the `-l c` 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.

![Diagram](image1.png)

---

**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.

![Diagram](image2.png)

---

**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()

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(), MsgKeyData_r()

MsgKeyData_r()  EOK is returned on success. This function does
\textbf{NOT} set \textit{errno}. If an error occurs, any value in the
Errors section may be returned.

\textbf{Errors:}

ESRCH   The thread indicated by \textit{rcvid} doesn’t exist.

EFAULT  A fault occurred when the kernel tried to access the
buffers provided.

\textbf{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);

pthread.create(&tid[1], NULL, server_dst_thread, NULL);
sleep(3);
    /* give time for channels to be created, sloppy but simple */
    
    /*
    * Send to server_dst_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;

November 2, 2006 C Library — M to O 1807
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));
    }

    return NULL;
}
} return NULL;
}

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:

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
### 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:**

(MsgReadv(), MsgReceive(), MsgReceivev(), MsgReply(),
MsgReplyv(), MsgWrite(), MsgWritev())
MsgReadv(), MsgReadv_r()

Read data from a message

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.
**MsgReadv(), MsgReadv_r()**

**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 `revid` 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

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
See also:

MsgRead(), MsgReceive(), MsgReceivev(), MsgReply(), MsgReplyv(),
MsgWrite(), MsgWritev()
Wait for a message or pulse on a channel

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:

```c
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 \textit{msg} buffer \textit{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 \texttt{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.

If you pass a non-NULL pointer for \texttt{info}, the functions store additional information about the message and the thread that sent it in the \texttt{.msg.info} structure that \texttt{info} points to. You can get this information later by calling \texttt{MsgInfo()}.

On success, \texttt{MsgReceive()} and \texttt{MsgReceive.r()} return:

\begin{itemize}
  \item \texttt{>0} A message was received; the returned value is a \texttt{rcvid} (receive identifier). You’ll use the \texttt{rcvid} with other \texttt{Msg*()} kernel calls to interact with and reply to the sending thread. \texttt{MsgReceive()} changes the state of the sending thread to \texttt{REPLY}-blocked when the message is received. When you use \texttt{MsgReply*()} to reply to the received message, the sending thread is made ready again. The \texttt{rcvid} encodes the sending thread’s ID and a local connection ID.
  \item \texttt{0} A pulse was received; \texttt{msg} contains a pulse message of type \texttt{.pulse}. When a pulse is received, the kernel space allocated to hold it is immediately released. The \texttt{.msg.info} structure isn’t updated.
\end{itemize}
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 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 `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 `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 `MsgSend()` on the client end) and the size of the message receive area on the server’s `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 `MsgReceive()` with a 1-Mbyte data area, then only the number of bytes determined by a network manager would in fact 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

<table>
<thead>
<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 maximum size for one-part message-pass is $2^{32} - 1$ (SSIZE_MAX).
See also:

ChannelCreate(), _msg_info, MsgBox(), MsgRead(), MsgReadv(),
MsgReceivePulse(), MsgReceivePulsev(), MsgReceivev(),
MsgReply(), MsgReplyv(), MsgSend(), MsgWrite(), MsgWritev(),
_pulse, TimerTimeout()
Synopsis:

```c
#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 ofEFAULT.

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.

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 (MsgReceivePulse) and (MsgReceivePulse_r) is the way they indicate errors. On success, they both return 0.

If an error occurred:

- (MsgReceivePulse) returns -1 and sets errno.
MsgReceivePulse(), MsgReceivePulse_r()

Software Systems GmbH & Co. KG.

- **MsgReceivePulse_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 `MsgReceivePulse()` 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 `MsgReceivePulse()` 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**

<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:**

`MsgDeliverEvent(), MsgReceive(), MsgReceivePulse(), MsgReceivev(), 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.

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()` returns -1 and sets `errno`.
- `MsgReceivePulsev_r()` returns the negative of a value from the Errors section. This function *doesn’t* set `errno`. 

1828  C Library — M to O  November 2, 2006
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

<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:

`MsgDeliverEvent()`, `MsgReceive()`, `MsgReceivePulse()`, `MsgReceivev()`, `MsgSendPulse()`, `_pulse`, `TimerTimeout()`
**Synopsis:**

```c
#include <sys/neutrino.h>

int MsgReceivev( int chid,
                 const iov_t * riow,
                 int rparts,
                 struct _msg_info * info);

int MsgReceivev_r( int chid,
                   const iov_t * riow,
                   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.

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.
MsgReceivev(), MsgReceivev_r() © 2006, QNX Software Systems GmbH & Co. KG.

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 message waiting.</td>
</tr>
</tbody>
</table>

Returns:

The only difference between \textit{MsgReceivev()} and \textit{MsgReceivev_r()} is the way they indicate errors. On success, both functions return a positive \textit{rcvid} if they received a message, or 0 if they received a pulse.

If an error occurs:

- \textit{MsgReceivev()} returns -1 and sets \textit{errno}.
- \textit{MsgReceivev_r()} returns the negative of a value from the Errors section. This function \textit{doesn’t} set \textit{errno}.

Errors:

- \texttt{EFAULT} A fault occurred when the kernel tried to access the buffers provided. Because the OS accesses the sender’s buffers only when \textit{MsgReceivev()} 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 EFAULT and \textit{MsgReceivev()} won’t unblock.
- \texttt{EINTR} The call was interrupted by a signal.
- \texttt{ESRCH} The channel indicated by \textit{chid} doesn’t exist.
- \texttt{ETIMEDOUT} A kernel timeout unblocked the call. See \textit{TimerTimeout()}. 

1832 C Library — M to O November 2, 2006
MsgReceivev(),
MsgReceivev_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:

ChannelCreate(), _msg_info, MsgInfo(), MsgRead(), MsgReadv(),
MsgReceive(), MsgReceivePulse(), MsgReceivePulsev(), MsgReply(),
MsgReplyv(), MsgWrite(), MsgWritev(), _pulse, TimerTimeout()
Reply with a message

Synopsis:

```
#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 REPL Y-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 `recv`d 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 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.

Blocking states

None. In the network case, lower priority threads may run.

Native networking

The `MsgReply()` function has increased latency when it’s used 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.

Returns:

The only difference between the `MsgReply()` and `MsgReply_.r()` functions is the way they indicate errors:

- `MsgReply()` If an error occurs, -1 is returned and `errno` is set.
- `MsgReply_.r()` 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 Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFAULT</td>
<td>A fault occurred in the sender’s address space when a server tried to access the sender’s return message buffers.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The thread indicated by rcvid doesn’t exist, or is no longer REPL Y-blocked on the channel, or the connection is detached.</td>
</tr>
<tr>
<td>ESRVRFault</td>
<td>A fault occurred when the kernel tried to access the buffers provided.</td>
</tr>
</tbody>
</table>

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>

Caveats:

The maximum size for one-part message-pass is $2^{32} - 1$ (SSIZE_MAX).

See also:

(MsgReceive(), MsgReceivev(), MsgReplyv(), MsgSend(), MsgSendv(), MsgWrite(), MsgWritev())
Synopsis:

```
#include <sys/neutrino.h>

int MsgReplyv( int rcvid,
               int status,
               const iov_t* rierv,
               int rparts );

int MsgReplyv_r( int rcvid,
                 int status,
                 const iov_t* rierv,
                 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.
- **riov**: An array of buffers that contains the message that you want to reply with.
- **size**: 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**

<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:

MsgReceive(), MsgReceivev(), MsgReply(), MsgSend(), MsgSendv(),
MsgWrite(), MsgWritev()
**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 libc` 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 REPL Y-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 REPL Y-blocked.

`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>
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:

**MsgSend()**
- **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()).

**MsgSend_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:**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The connection indicated by <code>coid</code> is no longer connected to a channel, or the connection indicated by <code>coid</code> doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The call was interrupted by a signal.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The server died while the calling thread was SEND-blocked or REPLY-blocked.</td>
</tr>
<tr>
<td>ESRVFAULT</td>
<td>A fault occurred in a server’s address space when accessing the server’s message buffers.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>A kernel timeout unblocked the call. See <code>TimerTimeout()</code></td>
</tr>
</tbody>
</table>

**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>
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()
Send a message to a channel (non-cancellation point)

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 REPL Y-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 REPL Y-blocked.

`MsgSend()` is a cancellation point for the `ThreadCancel()` kernel call; `MsgSendnc()` 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`. 
**MsgSendnc(), MsgSendnc_r()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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:**

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The connection indicated by <code>coid</code> is no longer connected to a channel, or the connection indicated by <code>coid</code> doesn’t exist. The channel may have been terminated by the server, or the network manager if it failed to respond to multiple polls.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The call was interrupted by a signal.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>The server died while the calling thread was SEND-blocked or REPLY-blocked.</td>
</tr>
<tr>
<td>ESRVRFault</td>
<td>A fault occurred in a server’s address space when accessing the server’s message buffers.</td>
</tr>
</tbody>
</table>
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>

**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()
Send a pulse to a process

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_MINA VAIL` through `PULSE_CODE_MAXA VAIL`.

- **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.

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.

**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.
$\textit{MsgSendPulse()}$, $\textit{MsgSendPulse\_r()}$

**EPERM**
This process doesn’t have sufficient permission to send a pulse to the connection, $\textit{coid}$.

**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>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 the server faults on delivery, the pulse is either lost or an error is returned.

**See also:**
Send a message to a channel

Synopsis:

```c
#include <sys/neutrino.h>

int MsgSendsv( int coid,
        const void* smsg,
        int sbytes,
        const iov_t* riiov,
        int rparts);

int MsgSendsv_r( int coid,
        const void* smsg,
        int sbytes,
        const iov_t* riiov,
        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.
- `riiov`  An array of buffers where the reply can be stored.
- `rparts`  The number of elements in the `riiov` 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 REPL Y-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 REPL Y-blocked.

`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`. 

November 2, 2006

C Library — M to O 1855
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.
ETIMEDOUT  A kernel timeout unblocked the call. See
TimerTimeout()

Classification:

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>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

ConnectAttach(), MsgReceive(), MsgReceivev(), MsgReply(),
MsgReplyv(), 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* riiov,
            int rparts );

int MsgSendsvnc_r( int coid,
            const void* smsg,
            int sbytes,
            const iov_t* riiov,
            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.
- `riiov` An array of buffers where the reply can be stored.
- `rparts` The number of elements in the `riiov` 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 REPL Y-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 REPL Y-blocked.

`MsgSendsv()` is a cancellation point for the `ThreadCancel()` kernel call; `MsgSendsvnc()` 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 `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

<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(), MsgSendnc(), MsgSendPulse(), MsgSendsv(), MsgSendv(), MsgSendvnc(), MsgSendvs(), MsgSendvsnc(), TimerTimeout()
**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 REPL Y-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 REPL Y-blocked.

It’s quite common to send two-part messages consisting of a fixed header and a buffer of data. The `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.

`MsgSendv()` is a cancellation point for the `ThreadCancel()` kernel call; `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_REPL Y. |
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 MsgSendv() and MsgSendv_r() functions is the way they indicate errors:

**MsgSendv()**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>The value of status from MsgReply*().</td>
</tr>
<tr>
<td>-1</td>
<td>An error occurred (errno is set), or the server called MsgError*() (errno is set to the error value passed to MsgError()).</td>
</tr>
</tbody>
</table>

**MsgSendv_r()**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>The value of status from MsgReply*().</td>
</tr>
<tr>
<td>negative value</td>
<td>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()).</td>
</tr>
</tbody>
</table>

Errors:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>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.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.</td>
</tr>
<tr>
<td>EINTR</td>
<td>The call was interrupted by a signal.</td>
</tr>
</tbody>
</table>
MsgSendv(), MsgSendv_r()

ESRCH The server died while the calling thread was SEND-blocked or REPL-y-blocked.

ESRVFAULT 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>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(), MsgReceive(), MsgReceivev(), MsgReply(), MsgReplyv(), MsgSend(), MsgSendnc(), MsgSendPulse(), MsgSendsv(), MsgSendsvnc(), MsgSendvnc(), MsgSendvs(), MsgSendvsnc(), TimerTimeout()
**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 REPL Y-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 REPL Y-blocked.

`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

<table>
<thead>
<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(), 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 libc` 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 REPL Y-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 REPL Y-blocked.

`MsgSendvs()` is a cancellation point for the `ThreadCancel()` kernel call; `MsgSendvsnc()` 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 \texttt{MsgSendvs()} and \texttt{MsgSendvs\_r()} functions is the way they indicate errors:

\begin{itemize}
\item \texttt{MsgSendvs()} \begin{itemize}
\item Success The value of \textit{status} from \texttt{MsgReply\_r()}. 
\item -1 An error occurred (\textit{errno} is set), or the server called \texttt{MsgError\_r()} (\textit{errno} is set to the error value passed to \texttt{MsgError()}).
\end{itemize}
\item \texttt{MsgSendvs\_r()} \begin{itemize}
\item Success The value of \textit{status} from \texttt{MsgReply\_r()}. 
\item negative value An error occurred (\textit{errno} is NOT set, the value is the negative of a value from the Errors section), or the server called \texttt{MsgError\_r()} (\textit{errno} is NOT set, the value is the negative of the error value passed to \texttt{MsgError()}).
\end{itemize}
\end{itemize}

Errors:

\begin{itemize}
\item \texttt{EBADF} The connection indicated by \textit{coid} is no longer connected to a channel, or the connection indicated by \textit{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.
\item \texttt{EFAULT} A fault occurred when the kernel tried to access the buffers provided. This may have occurred on the receive or the reply.
\item \texttt{EINTR} The call was interrupted by a signal.
\item \texttt{ESRCH} The server died while the calling thread was SEND-blocked or REPLY-blocked.
\item \texttt{ESRVRFault} A fault occurred in a server’s address space when accessing the server’s message buffers.
\end{itemize}
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()`, `MsgSendnc()`, `MsgSendPulse()`, `MsgSendsv()`, `MsgSendsvnc()`, `MsgSendv()`, `MsgSendvnc()`, `MsgSendvsnc()`, `TimerTimeout()`
**MsgSendvsnc(), MsgSendvsnc_r()**

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 REPL Y-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 REPL Y-blocked.

`MsgSendvs()` is a cancellation point for the `ThreadCancel()` kernel call; `MsgSendvsnc()` 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 `MsgSendvsnc()` and `MsgSendvsnc_r()` functions is the way they indicate errors:

**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()`).

**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.
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()`, `MsgSendnc()`, `MsgSendPulse()`,
`MsgSendsv()`, `MsgSendsvnc()`, `MsgSendv()`, `MsgSendvnc()`,
`MsgSendvs()`, `TimerTimeout()`
**MsgVerifyEvent(), MsgVerifyEvent_r()**

Check the validity of a receive ID and an event configuration

**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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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()`, `sigevent`
**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
**MsgWrite(), MsgWrite_r()**

**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:

ChannelCreate(), MsgRead(), MsgReadv(), MsgReceive(),
MsgReceivev(), MsgReply(), MsgReplyv(), MsgWritev()
Write a reply

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:

`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 REPL Y-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 `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 `MsgRead*()` to read messages in small pieces, and use `MsgWritev()` 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 `MsgWritev()` followed by a call to `MsgReply*()`.

**Blocking states**

None. In the network case, lower priority threads may run.

**Returns:**

The only difference between the `MsgWritev()` and `MsgWritev_r()` functions is the way they indicate errors:

- `MsgWritev()` The number of bytes written. If an error occurs, -1 is returned and `errno` is set.

- `MsgWritev_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

<table>
<thead>
<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_CACHEONLY** — a QNX Neutrino extension. See the Caveats section.
  - **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.

This flag was added in the QNX Neutrino Core OS 6.3.2.
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_atime` 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
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 S1O

**Safety**

<table>
<thead>
<tr>
<th>Description</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>
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

<table>
<thead>
<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:

mlock(), munlock(), mlockall()
munmap()
Unmap previously mapped addresses

Synopsis:

```c
#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:

EINV    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.

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()
Synopsis:

```c
#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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_io(), munmap()
munmap_device_memory()

Unmap previously mapped addresses

Synopsis:

```c
#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 unmapped.
- `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`
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

<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_device_memory()`, `munmap()`, `munmap_device_io()`
munmap_flags()

Unmap previously mapped addresses exercising more control

Synopsis:

```c
#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 `-memmgr_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:

- EINV  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

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(), mprotect(), munmap(), shm_open(), shm_unlink()
**name_attach()**  
© 2006, QNX Software Systems GmbH & Co. KG.

Register a name in the namespace and create a channel

**Synopsis:**
```
#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?
             */
          }
    }
```c
/* 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;
        }
    }
```
/* 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>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>
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:

```
#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()`.

November 2, 2006
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()
name_detach()
Remove a name from the namespace and destroy the channel

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>
</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:

ChannelDestroy(), dispatch*() functions, name_attach(), 
name_close(), name_open(), resmgr_detach()
name_open()

Open a name for a server connection

Synopsis:
#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()`
nanosleep()
Suspend a thread until a timeout or signal occurs

Synopsis:
#include <time.h>

int nanosleep( const struct timespec* rqtp,
               struct timespec* rmtp );

Arguments:

rqtp A pointer to a timespec structure that specifies the time interval for which you want to suspend the thread.

rmtp 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.
**nanosleep()**

**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

**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`, `clock_gettime()`, `clock_settime()`, `sleep()`, `timer_create()`, `timer_delete()`, `timer_gettime()`, `timer_settime()`, `timespec`
**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.

- **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()`.

**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>Read the Caveats</td>
<td>Yes</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:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A too-high rate of interrupts occurred during the calibration routine.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The process doesn’t have superuser capabilities.</td>
</tr>
</tbody>
</table>

**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()**
© 2006, QNX Software Systems GmbH & Co. KG.

**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 `-l c` 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>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Yes</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>

**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.
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

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<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()`, `nanospin_calibrate()`, `nanospin_count()`, `nanospin_ns_to_count()`, `sched_yield()`, `sleep()`, `timer_create()`
**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).
Errors:

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_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

<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>
**nanospin_ns_to_count()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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>
</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:

delay(), napms()
napms()

Sleep for a given number of milliseconds

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

<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(), nap()
Synopsis:

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

int nbaconnect( int s,
                const struct sockaddr * name,
                size_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 `nbaconnect()` function is called in place of `connect()`, to prevent a nonblocking `connect()` from blocking during an autoconnect (see `/etc/autoconnect`).

When the autoconnect behavior is used, `connect()` may block your application while waiting for the autoconnect to complete; `nbaconnect()` allows your application to continue executing during the autoconnect.

The `nbaconnect()` function takes the same arguments as `connect()`, but it differs in the return value when an autoconnect is required. If an autoconnect is required, a file descriptor (`fd`) is returned. The `fd` is used in the call to `nbaconnect_result()` to get the `errno` related to the autoconnect and the connect attempt.
Since `nbaconnect_result()` is a blocking call, it’s recommended that you call `select()` first to determine if there’s data available on the pipe.

When the data’s available, call `nbaconnect_result()` to get the status of the `nbaconnect()` attempt.

If an autoconnect isn’t required, `nbaconnect()` returns -1 and exhibits the same behavior as `connect()` on nonblocking sockets (e.g. if -1 is returned and `errno` is set to EINPROGRESS, it’s possible to do a `select()` for completion by selecting the socket for writing).

**Returns:**

A file descriptor that you can pass to `nbaconnect_result()` to get the result of the `nbaconnect()` attempt, or -1 if an error occurred (`errno` is set).

**Errors:**

Any value from the Errors section in `connect()`, as well as:

- **EINVAL** The socket file descriptor being passed isn’t nonblocking.

**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:

The `pipe` manager must be available.

See also:

`accept()`, `bind()`, `connect()`, `errno`, `fcntl()`, `getsockname()`,
`nbacconnect_result()`, `open()`, `pipe()`, `read()`, `select()`, `socket()`, `write()`

`/etc/autoconnect.pipe` in the Utilities Reference.
**Synopsis:**

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

int nbacconnect_result( int fd,
    int * error );
```

**Arguments:**

- `fd` The file descriptor returned by `nbacconnect()`.
- `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 `nbacconnect_result()` function gets the status of the previous `nbacconnect()` call when an `fd` was returned. Since `nbacconnect_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()`, `nbaconnect()`, `select()`

`autoconnect` in the Utilities Reference.
ND_NODE_CMP()  © 2006, QNX Software Systems GmbH & Co. KG.

Compare two node descriptors

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_ntostr(), netmgr_remoteln(), netmgr_strtond()

Qnet Networking chapter of the Programmer’s Guide
Qnet Networking chapter of the System Architecture guide
Synopsis:

```
#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()`, `getbyname()`, `getnetent()`, `setnetent()`

`/etc/networks` in the Utilities Reference
Synopsis:

```
#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:`
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 an `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:

```
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:

```
netmgr_ndtostr(ND2S_SEP_FORCE | ND2S_DIR_HIDE | ND2S_NAME_HIDE | ND2S_DOMAIN_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;
    }
```
```c
}  
else {  
    printf ("Node id for %s, relative to %s, is %d.\n",  
            path1, path2, nd3);  
}  

len = netmgr_ndtostr ( ND2S_DIR_HIDE | ND2S_DOMAIN_SHOW |  
    ND2S_NAME_SHOW | ND2S_QOS_SHOW, nd1, buff, 100 );  
if (len == -1) {  
    perror ("netmgr_ndtostr" );  
}  
else {  
    printf ("Node name for %d is %s.\n", nd1, buff);  
    return EXIT_FAILURE;  
}  

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>

**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
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:

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_ndtostr()`.

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_ndtostr()`, `netmgr_remote_uid()`

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:

- `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`.

<table>
<thead>
<tr>
<th>If:</th>
<th><code>nextafter()</code> returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y &lt; x</code></td>
<td>The next possible floating-point value less than <code>y</code></td>
</tr>
<tr>
<td><code>y &gt; x</code></td>
<td>The next possible floating-point value greater than <code>x</code></td>
</tr>
</tbody>
</table>

continued…
If:

| $x$ is NAN | NAN |
| $y$ is NAN | NAN |
| $x$ is finite | $\pm$HUGE_VAL, according to the sign of $x$ ($\text{errno}$ is set) |

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.

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",
            (uint32_t)(*((uint64_t*)&d) >> 32),
            (uint32_t)(*((uint64_t*)&d)));
}

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:
Next possible value before 0.000000 is 0.000000
-->0x00000000 00000000
Next possible value after 0.000000 is 0.000000
-->0x00000000 00000001

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>
**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, \texttt{nftw()} only reports files on the same filesystem as \texttt{path}.

FTW_PHYS If set, \texttt{nftw()} performs a physical walk and doesn’t follow any symbolic link.

\textbf{Library:}

\texttt{libc}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

\begin{itemize}
\item These functions are in \texttt{libc.a}, but not in \texttt{libc.so} (in order to save space).
\end{itemize}

\textbf{Description:}

The \texttt{nftw()} function recursively descends the directory hierarchy identified by \texttt{path}. For each object in the hierarchy, \texttt{nftw()} calls the user-defined function \texttt{fn()}, passing to it:

\begin{itemize}
\item a pointer to a NULL-terminated character string containing the name of the object
\item a pointer to a \texttt{stat} structure (see \texttt{stat()}) containing information about the object
\item an integer. Possible values of the integer, defined in the \texttt{<nftw.h>} header, are:
\begin{itemize}
\item \texttt{FTW_F} The object is a file.
\item \texttt{FTW_D} The object is a directory.
\item \texttt{FTW_DNR} The object is a directory that can’t be read. Descendents of the directory aren’t processed.
\item \texttt{FTW_DP} The object is a directory, and its contents have been reported. See the FTW_DP flag above.
\end{itemize}
\end{itemize}
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).
nftw(), nftw64()

Classification:

nftw() is POSIX 1003.1 XSI; nftw64() 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>

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 fn() 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 fn() return a nonzero value at its next invocation.

See also:

ftw(), longjmp(), malloc(), stat()
nice() Change the priority of a process

Synopsis:

```c
#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.
nice()

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

<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:

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:

execl(), execl(), execvp(), execvpe(), execve(), execvp(), execvp()


**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...*
**nrand48()**

<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:**

*drand48(), erand48(), jrand48(), lcong48(), lrand48(), mrand48(), seed48(), srand48()*
Synopsis:

```
#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>
</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>

November 2, 2006
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 ntohl() 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>
</tbody>
</table>

continued…
ntohl()  

Safety

| Signal handler | Yes |
| Thread         | Yes |

See also:

gethostbyname(), getservent(), htonl(), htons(), ntohl()
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

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>
ntohs()

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:

gethostbyname(), getservent(), htonl(), htons(), ntohl()
offsetof()

Return the offset of an element within a structure

Synopsis:

#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:

#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>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>

**Caveats:**

This is a macro.
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.
**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
open(), open64()

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 open64()).

EEXIST The O_CREAT and O_EXCL flags are set, and the named file exists.

EINVAL 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.
The open() function isn’t implemented for the filesystem specified in path.

A component of the path prefix isn’t a directory.

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).

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.

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:

```c
#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

<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 open() function includes POSIX 1003.1-1996 and QNX extensions.

**See also:**

chmod(), close(), creat(), dup(), dup2(), errno, fcntl(), fstat(), lseek(), read(), write()
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 {
        for(;;) {
            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()
**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:**

```c
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

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:

dup(), sopenfd()
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.
openlog()

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.
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
openpty()

Find an available pseudo-tty

Synopsis:

```c
#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

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:

fork(), forkpty(), login_tty(), termios
**Synopsis:**
```
#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>
<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>
**out8()**

**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()`, `out8s()`, `out16()`, `out16s()`, `out32()`, `out32s()`
out8s()
Write 8-bit values to a port

Synopsis:

#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
out8s()

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.

See also:

`in8()`, `in8s()`, `in16()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_io()`, `out8()`, `out16()`, `out16s()`, `out32()`, `out32s()`
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

<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()`, `out32()`, `out32s()`
out16s()

Write words to a port

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
out16s()

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:

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()
out32(), outbe32(), outle32()  © 2006, QNX Software Systems GmbH & Co. KG.

Write a 32-bit value to a port

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>Cancelation 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()`
out32s()

Write longs to a port

Synopsis:

#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
**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.

**See also:**

`in8()`, `in8s()`, `in16()`, `in16s()`, `in32()`, `in32s()`, `mmap_device_io()`, `out8()`, `out8s()`, `out16()`, `out16s()`, `out32()`
C Library — P to R
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>
pathconf()

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).

_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 root 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
  will generate an error.

_PC_VDISABLE
  If defined (not -1), this is the character value which can be used
  to individually disable special control characters in the
  termios control structure.

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.
pathconf()

ELOOP Too many levels of symbolic links or prefixes.

ENAMETOOLONG
The path argument, or a component of path, is too long.

ENOENT The file doesn’t exist.

ENOSYS The pathconf() function isn’t implemented for the filesystem specified in path.

ENOTDIR A component of the path prefix isn’t a directory.

Examples:
```c
#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>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:

chown(), confstr(), errno, fpathconf(), sysconf(), termios

getconf in the Utilities Reference

Understanding System Limits chapter of the Neutrino User’s Guide
**Synopsis:**

```c
#include <libgen.h>

char *pathfind( const char *path,
    const char *name,
    const char *mode );

char *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.
pathfind(), pathfind_r()

k  Sticky bit.

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
pathmgr_symlink()
Create a symlink

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

<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:

`pathmgr_unlink()`, `symlink()`, `unlink()`
**Synopsis:**

```c
#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>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>continued…</td>
</tr>
</tbody>
</table>

© 2006, 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>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`pathmgr_unlink()`, `symlink()`, `unlink()`
**pause()**

*Suspend the calling thread until delivery of a signal*

**Synopsis:**

```c
#include <unistd.h>

int pause( void );
```

**Library:**

```c
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>
</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(), errno, sigaction()
**pccard_arm()**

Arm the devp-pccard server

**Synopsis:**

```c
#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_DEVVIDEO` — any video adapter.
- `event` The type of event that you want to be notified of:
  - `PCCARD_ARM_INSERT_REMOVE` — card insertion/removal.
- `coid` A connection ID, obtained from `ConnectAttach()`, that’s used to send the pulse.
**pccard.arm()**

**Library:**

```c
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)
{
    pccard_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);
    }
```

November 2, 2006
if ((chid = ChannelCreate (_NTO.CHANNEL, _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_INSERTREMOVE, 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);
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>

See also:

pccard_attach(), pccard_detach(), pccard_info(), pccard_lock(),
pccard_raw_read(), pccard_unlock()
**pccard_attach()**

Attach to the **devp-pccard** server

**Synopsis:**

```c
#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()`
**pccard_detach()**

`Detach from the devp-pccard server`

**Synopsis:**

```c
#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

<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_info()`, `pccard_lock()`, `pccard_raw_read()`, `pccard_unlock()`
Obtain socket information from the \texttt{devp-pccard} server

**Synopsis:**

\begin{verbatim}
#include <sys/pccard.h>

int pccard_info( pccard_t handle, 
    int socket, 
    struct _pccard_info* info, 
    unsigned size );
\end{verbatim}

**Arguments:**

- \textit{handle} \quad The handle returned by \texttt{pccard_attach}.
- \textit{socket} \quad 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.
- \textit{info} \quad A pointer to a \texttt{_pccard_info} structure that the function fills with the socket information. For more information, see below.
- \textit{size} \quad Size of the \texttt{_pccard_info} structure.

**Library:**

\texttt{libpccard}

Use the \texttt{-l pccard} option to \texttt{qcc} to link against this library.

**Description:**

The \texttt{pccard_info()} function call retrieves socket setup information from the \texttt{devp-pccard} server. The information is returned in the \texttt{_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)
    uint32_t flags;    // Flags (_PCCARD_FLAG_*)
    uint8_t vcc;       // Current Vcc (in tenths of volts)
    uint8_t vpp;       // Current Vpp (in tenths of volts)
    uint8_t num_windows; // Number of windows described below
    uint8_t index;     // Index for CardBus devices
    uint16_t manufacturer; // Manufacturer ID from PCCARD
    uint16_t card_type; // Card Type from PCCARD
    uint16_t device_id; // CardBus device id
    uint16_t vendor_id; // CardBus vendor id
    uint16_t busnum;    // PCI bus number
    uint16_t devfuncnum; // PCI device and function number
    struct pccard_window {
        uint16_t window;  // Window type (_PCCARD_WINDOW_*)
        uint16_t flags;    // Window flags (_PCCARD_WINFLAG_*)
        mpid_t pid;       // Locking pid
        uint16_t device;   // Device type (_PCCARD_DEV_*)
        uint16_t dummy;
        uint32_t dev_size; // Size of memory device
        uint32_t reserved3;
        union {
            struct pccard_irq {
                uint16_t flags; // (_PCCARD_IRQFLAG_*)
                uint16_t irq;
            }
            struct pccard_memio {
                uint32_t base; // Base address (in host address space)
                uint32_t size; // Size of window
                uint32_t offset; // offset of region from base of card
                uint16_t flags; // (_PCCARD_MEMIOFLAG_*)
                uint16_t dummy2;
            }
        } un;
    } window[_PCCARD_MAX_WINDOWS];
};
```

Returns:

A positive integer

Success. The socket parameter is returned.

-1 An error occurred (errno is set).
Errors:

ENODEV  Invalid socket 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_lock(),
pccard_raw_read(), pccard_unlock()
Synopsis:

```
#include <sys/pccard.h>

int pccard_lock( pccard_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

<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_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.
**pccard_raw_read()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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>
<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_unlock()*


**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:**

`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).
**pccard_unlock()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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

**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:**

- `pccard_arm()`, `pccard_attach()`, `pccard_detach()`, `pccard_info()`, `pccard_lock()`, `pccard_raw_read()`
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

<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_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:

- **libc**: Use the `-l c` 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 `rsrdbmgr*` 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 \texttt{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 \texttt{pci_dev_info} structure are all physical addresses.

This structure has the following members:

\begin{verbatim}
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. \texttt{PCI_CLASS_DISPLAY | PCI_SUBCLASS_DISPLAY_XGA}).
\end{verbatim}


**pci_attach_device()**

```
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).

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.
```
The `pci_attach_device()` function attaches a PCI device to the system.

```c
uint32_t BaseAddressSize [6]
```

The size of the base address aperture into the board (an array of six `uint32_t` items).

```c
uint64_t PciRom
```

The PCI ROM address.

```c
uint64_t CpuRom
```

The CPU ROM address.

```c
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**: Vendor ID
- **PCI_SEARCH_VENDEV**: Device ID and Vendor ID
- **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     Iq
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_IO_ADDR( 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.
EINVVAL 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:
#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 )
```c
pci_attach_device()
{
    int pidx;
    void* hdl;
    void* retval;
    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**

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

*continued...*
pci_attach_device()

Safety

| Thread | Yes |

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()
Synopsis:

```c
#include <hw/pzi.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`. 
pci_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>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()
Synopsis:

```
#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.
pci_find_class()

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

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_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_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 n'th 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.
**pci_find_device()**

© 2006, QNX Software Systems GmbH & Co. KG.

**Description:**

The *pci_find_device()* function returns the location of the *n*th 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

**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_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_irq_routing_options() Retrieve PCI IRQ routing information

Synopsis:
#include <hw/pci.h>

int pci_irq_routing_options(
    IRQRoutingOptionsBuffer * buf,
    uint32_t * irq);

Arguments:

buf  A pointer to a 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/.

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 dabuf [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.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 Irq Map = %x\n", irq);
  pci_detach (phdl);
```
```c
return (0);
}
```

**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_read_config32()`, `pci_rescan_bus()`, `pci_write_config()`, `pci_write_config8()`, `pci_write_config16()`, `pci_write_config32()`
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.
 pci_map_irq()

Returns:

 PCI_SUCCESS
    Success.
 PCI_SET_FAILED
    The PCI server was unable to map the `intn/intpin`.
 PCI_UNSUPPORTED_FUNCTION
    This function isn’t supported.
 -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_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

<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_present()

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

Synopsis:

```c
#include <hw/pci.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()
PCI Read Configuration Space

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.
pci_read_config8()

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(),

2072 C Library — P to R November 2, 2006
pci_read_config8()
**pci_read_config16()**  
© 2006, QNX Software Systems GmbH & Co. KG.

*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, \ldots, 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.
pci_read_config16()

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

<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_config32()`, `pci_rescan_bus()`,
pci_read_config16()

pci_write_config(), pci_write_config8(), pci_write_config16(),
pci_write_config32()
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()`
pci_rescan_bus()
Rescan the PCI bus for added or removed devices

Synopsis:

```c
#include <hw/pci.h>

int pci_rescan_bus( void );
```

Library:

```
libc
```

Use the `-l c` 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>
</table>

continued...
**pci_rescan_bus()**

© 2006, QNX Software Systems GmbH & Co. KG.

<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:

`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_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.

**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

**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_present(), pci_read_config(),
pci_read_config8(), 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:

```c
#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

**Safety**

<table>
<thead>
<tr>
<th>Category</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>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_read_config16()`, `pci_read_config32()`, `pci_rescan_bus()`, `pci_write_config()`, `pci_write_config16()`, `pci_write_config32()`
pci_write_config16()

Write 16-bit values to the configuration space of a device

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 \texttt{pci\_attach()} before calling any of the other PCI functions.

\textbf{Returns:}

\begin{itemize}
    \item [\texttt{PCI\_BAD\_REGISTER\_NUMBER}] An invalid register offset was given.
    \item [\texttt{PCI\_BUFFER\_TOO\_SMALL}] The \textit{size} argument is greater than 50 words.
    \item [\texttt{PCI\_SUCCESS}] The device or function was found.
    \item [-1] You haven’t called \texttt{pci\_attach()}, or the call to it failed.
\end{itemize}

\textbf{Classification:}

QNX Neutrino

\textbf{Safety}

\begin{tabular}{l|c}
\textbf{Cancellation point} & \textbf{Yes} \\
\textbf{Interrupt handler} & \textbf{No} \\
\textbf{Signal handler} & \textbf{Yes} \\
\textbf{Thread} & \textbf{Yes}
\end{tabular}

\textbf{See also:}

\texttt{pci\_attach()}, \texttt{pci\_attach\_device()}, \texttt{pci\_detach()}, \texttt{pci\_detach\_device()},
\texttt{pci\_find\_class()}, \texttt{pci\_find\_device()}, \texttt{pci\_present()}, \texttt{pci\_read\_config()},
\texttt{pci\_read\_config8()}, \texttt{pci\_read\_config16()}, \texttt{pci\_read\_config32()},
\texttt{pci\_rescan\_bus()}, \texttt{pci\_write\_config()}, \texttt{pci\_write\_config8()},
\texttt{pci\_write\_config32()}
PCI Write Configuration Space

**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.
pci_write_config32()

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

**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()`
**Synopsis:**

```
#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.

This function is in `libc.a`, but not in `libc.so` (in order to save space).

**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>
<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()`, `stderr`, `strerror()`
Create a pipe

**Synopsis:**

```c
#include <unistd.h>

int pipe( int *fdes[2] );
```

**Arguments:**

- `fdes` 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 `fdes[0]`, and a file descriptor for the write end of the pipe in `fdes[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 `fdes[1]` and read it from file descriptor `fdes[0]`. If you read from file descriptor `fdes[0]`, it returns the data written to `fdes[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_ftime`, `st_ctime`, `st_atime` and `st_mtime` fields of the pipe for updating.

**Returns:**

0  Success.

-1  An error occurred (`errno` is set).

**Errors:**

EMFILE  The calling process doesn’t have at least 2 unused file descriptors available.

ENFILE  The number of simultaneously open files in the system would exceed the configured limit.

ENOSPC  There’s insufficient space available to allocate the pipe buffer.

ENOSYS  There’s no pipe manager running.

EROFS  The pipe pathname space is a read-only filesystem.

**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>

© 2006, QNX Software Systems GmbH & Co. KG.
See also:

errno, fcntl(), nbaconnect(), open(), popen(), read(), write()

pipe in the Utilities Reference
poll()

Multiplex input/output over a set of file descriptors

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 `-l c` 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:

```c
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:

- **POLLERR**: An error has occurred on the device. This flag is valid only in the `revents` bitmask; it’s ignored in the `events` member.
- **POLLHUP**: 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 `POLLPRI`, `POLLRDNORM`, `POLLRDBAND`, or `POLLWRBAND` are not mutually exclusive. If the remote end of a socket is closed, `poll()` indicates a `POLLPRI` event rather than `POLLHUP`. This flag is valid only in the `revents` bitmask; it’s ignored in the `events` member.
- **POLLPRI**: High-priority data may be read without blocking.
- **POLLRDBAND**: Priority data may be read without blocking.
- **POLLRDNORM**: Normal data may be read without blocking.
- **POLLWRBAND**: Priority data may be written.
POLLWRNORM

Equivalent to POLLOUT.

The significance and semantics of normal, priority, and high-priority data are file- and device-specific.

If the value of `fd` is less than 0, events are ignored; and `revents` are set to 0 in that entry on return from `poll()`.

In each `pollfd` structure, `poll()` clears the `revents` member, except that where the application requested a report on a condition by setting one of the bits of events listed above, `poll()` sets the corresponding bit in `revents` if the requested condition is true. In addition, `poll()` sets the POLLHUP, POLLERR, and POLLNVAL flag in `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, `poll()` waits at least `timeout` milliseconds for an event to occur on any of the selected file descriptors. If the value of `timeout` is 0, `poll()` returns immediately. If the value of `timeout` is -1, `poll()` blocks until a requested event occurs or until the call is interrupted.

The `poll()` function isn’t affected by the O_NONBLOCK flag.

The `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.

Returns:

- `> 0` Total number of file descriptors that have been selected.
- `0` The call timed out, and no file descriptor has been selected.
- `-1` Failure, and `errno` is set.
Errors:

EAGAIN The allocation of internal data structures failed but a subsequent request may succeed.

EINTR A signal was caught during poll()

EFAULT The 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)
```

© 2006, QNX Software Systems GmbH & Co. KG.
{  
    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);  
    }
```c
} 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

<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>

Caveats:

Not all managers support POLLPRI, POLLPRI, POLLERR, and POLLHUP.
See also:

read(), select(), write()

<sys/poll.h>
**popen()**

Execute a command, creating a pipe to it

**Synopsis:**

```c
#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:

```c
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 The `mode` argument is invalid.

ENOSYS There’s no pipe manager running.

The `popen()` function may also set `errno` values as described by the `pipe()` and `spawnl()` functions.
Examples:

```c
/*
 * 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

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>No</td>
</tr>
</tbody>
</table>

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`.  

posix_mem_offset(), posix_mem_offset64()

Classification:

posix_mem_offset() is POSIX 1003.1 TYM; posix_mem_offset64() is
for 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:

posix_typed_mem_get_info(), posix_typed_mem_open()
posix_memalign()
Allocate aligned memory

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(size_t *)`.
- `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`. 
posix_memalign()

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

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, free(), malloc(), memalign()
**Synopsis:**

```c
#include <sys/mman.h>

int posix_typed_mem_get_info(int fildes,
                             struct posix_typed_mem_info *info);
```

**Arguments:**

- `fildes`: Represent a typed memory object.
- `info`: Pointer to `posix_typed_mem_info` structure that contain `posix_tmi_length` field.

**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 for obtaining the amount of memory currently available.

The `posix_typed_mem_get_info()` function allows an application to immediately determine 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 shall return, 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 `posix_typed_mem_get_info()` function returns the maximum length which may be successfully allocated by the typed memory object designated by `fildes`. 
The maximum length is dynamic; it’s available in the `posix_tmi_length` field, pointed to by `info`, of the `posix_typed_mem_info` structure. The maximum length is 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` is 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 unspecified.

This function was added in the QNX Neutrino Core OS 6.3.2.

**Returns:**

0 Success.

Errors as mentioned below

An error has occurred.

**Errors:**

EBADF The `fildes` argument isn’t a valid open file descriptor.

ENODEV The `fildes` isn’t connected to a memory object supported by this function.

**Classification:**

`posix_typed_mem_get_info()` is POSIX 1003.1-2001

**Safety**

Cancellation point  No

continued...
posix_typed_mem_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>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`posix_mem_offset()`, `posix_typed_mem_open()`
posix_typed_mem_open() © 2006, QNX Software Systems GmbH & Co. KG.
Open a typed memory object

Synopsis:

```c
#include <sys/mman.h>

int posix_typed_mem_open(const char * name,
                          int oflag,
                          intt * tflag);
```

Arguments:

- `name`: Point to the string that specifies the typed memory object.
- `oflag`: Flag of the open file description that determines the file status.
- `tflag`: Flag that determines how the typed memory object behaves when mapped.

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_open()` function establishes a connection between the typed memory object specified by the string pointed to by `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.

Set the file status flags of the open file description according to the value of `oflag` as specified below:

- `O_RDONLY` Open for read access only.
posix_TYPED_MEM_OPEN()

O_WONLY     Open for write access only.
O_RDWR      Open for read or write access.

The value of tflag determine how the typed memory object behaves when subsequently mapped by calls to mmap(). It can take one of the following values:

POSIX_TYPED_MEM_ALLOCATE
   Allocate on mmap().

POSIX_TYPED_MEM_ALLOCATE_CONTIG
   Allocate contiguously on mmap().

POSIX_TYPED_MEM_MAP_ALLOCATABLE
   Map on mmap() that doesn’t affect allocation.

If name begins with the slash character, then processes calling posix_TYPED_MEM_OPEN() with the same value of name shall refer to the same typed memory object. If name doesn’t begin with the slash character, the effect is implementation-defined (?). The interpretation of slash characters other than the leading slash character in name is implementation-defined (?).

When tflag is set to POSIX_TYPED_MEM_ALLOCATE, any subsequent call to mmap() that uses the returned file descriptor shall result in 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) non-contiguous 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 shall result in 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` is not set to either POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG, any subsequent call to `mmap()` that uses the returned file descriptor shall map 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 specified, any subsequent call to `mmap()` using the returned file descriptor shall map 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 shall be 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 unspecified when passed a file descriptor connected to a typed memory object by this function. The file status flags of the open file description shall be set according to the value of `oflag`. Applications shall specify exactly one of the three access mode values described below and defined in the header, as the value of `oflag`.

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.

   Success.
-1  An error has occurred (errno is set).

Errors:

EACCES  The typed memory object exists and the permissions specified by oflag are denied.
EINTR  The posix_typed_mem_open() API is interrupted by a signal.
EINVAL  The flags specified in tflag are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, 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.
ENAMETOOLONG  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_typed_mem_open() is POSIX 1003.1-2001
### 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_mem_offset()*
**Synopsis:**

```
#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 ≤ 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( "%f\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()`
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:

```
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()`.
**pread(), pread64()**

**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 fildes.

**Classification:**

preat() is POSIX 1003.1 XSI; pread64() 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>
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.
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.
printf()

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 wint_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.

- **l** 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.
Convert an argument of type `double` into a decimal notation in the form \([-\] \textit{d} \textit{ddd} \textit{e} [+|−] \textit{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.

Convert an argument of type `double` into a decimal notation in the form \([-\] \textit{d} \textit{dd}.\textit{ddd} \textit{e} [+|−] \textit{dd}\) 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 \([-\] \textit{inf} or \([-\] \textit{nan}. The `F` specifier produces \([-\] \textit{INF} or \([-\] \textit{NAN}.}
\texttt{g, G } Convert an argument of type \textbf{double} using either the \texttt{e} or \texttt{f} (or \texttt{E}, for a \textbf{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 \texttt{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 \texttt{f} or \texttt{F} conversion specifiers.

\texttt{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.

\texttt{o } Convert an argument of type \textbf{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.

\texttt{p } Convert an argument of type \textbf{void *} into a value of type \textbf{int}, and format the value as for a hexadecimal (\texttt{x}) conversion.

\texttt{s } Write the characters from the string specified by an argument of type \textbf{char *}, up to, but not including the terminating NUL character (‘\texttt{\0}’), to the output stream. If you specify a precision, no more than that many characters are written.

If you use an \texttt{l} (“el”) qualifier, the argument is interpreted as a pointer to a \textbf{wchar_t} array, and each wide character, including the terminating NUL, is converted as if by a call to \texttt{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.

\texttt{u } Convert an argument of type \textbf{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.

`x, X` 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 x or X conversions, respectively, as the hexadecimal digits. Subject to the alternate-form control flag, 0x or 0X is prepended to the output.

`%` 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 inf or -inf for infinity, and nan or -nan for NANS.

For example, a specifier of the form `%8.*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.

**Returns:**

The number of characters written, excluding the terminating NULL, or a negative number if an error occurred (errno is set).

**Examples:**

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

int main( void )
{

```
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:**

```c
#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:**

```c
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:**
`daemon()`, `exit()`, `procmgr_event_notify()`, `procmgr_event_trigger()`, `procmgr_guardian()`, `procmgr_session()`
procmgr_event_notify() 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.

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_SYNC` — notify the caller of any calls to `sync()` the filesystems.

This flag was added in the QNX Momentics 6.3.0 SP2.
### procmgr_event_notify()

Setting `flags` to 0 (zero) unarms the event.

**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>
```

November 2, 2006
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <sys/procs.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 );
    }
}

2140  C Library — P to R  November 2, 2006
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;
}

/*
 * check_if_running - This will walk through all processes
 * to see if this particular one is still running.
 */

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);
                    }
                }
            }
        }
    }

    closedir (dirp);
}
/* You should close fd to prevent memory leaking */
close(fd);
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
**Synopsis:**

```
#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:**

```
#include <sys/procmgr.h>

int main ( void )
{
    procmgr_event_trigger( PROCMGR_EVENT_SYNC );
}
```
**procmgr_event_trigger()**

© 2006, 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:

`procmgr_event_notify()`, `sync()`
procmgr_guardian()

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:

```plaintext
A (parent)           A dies
├── B
└── C (guardian)    └── B
```

Specifying a guardian for child processes.
procmgr_guardian()

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 {
      /*
       * This is the guardian...
       */
      /* Wait for a child to die */
      while(waitpid(child, 0, 0) != child);
      /* Then become the child again... */
      if((child = fork()) == 0) {
        pid_t parent = getppid();
        /* Wait for parent to die... */
        fprintf(stderr, "child %d waiting on parent %d\n",
                getpid(), parent);
        while(waitpid(parent, 0, 0) != parent);
        /* Then run a new child */
        if((child = fork()) == 0) {
          printf(stderr, "New child %d\n",
                  getpid());
        } else {
          /* Become the child again... */
          if(child == -1) {
            fprintf(stderr, "Unable to start child\n");
          } else {
            /* Print messages */
            printf(stderr, "I am child %d\n",
                    getpid());
            while(1);
          }
        }
      }
    }
  }
}
```

2146 C Library — P to R November 2, 2006
/* 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:

```
#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

<table>
<thead>
<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:

setsid(), tcsetpgrp(), tcsetsid()
Synopsis:

```
char * __progname
```

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()`, `getprotobyname()`, `getprotobynumber()`, `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

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_cancel(), pthread_detach(), pthread_exit(), ThreadDestroy()
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:

- EOK: Success.
- ENOMEM: Insufficient memory to record fork handlers.

Classification:

POSIX 1003.1 THR

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:

`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.
**pthread_attr_destroy()**

**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()**  © 2006, QNX Software Systems GmbH & Co.

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()

**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:**

`pthread_attr_setdetachstate()`, `pthread_attr_init()`, `pthread_create()`.
**pthread_attr_getguardsize()** © 2006, 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

<table>
<thead>
<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_setguardsize()

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>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_attr_setinheritsched(), pthread_attr_init(), pthread_create()
Get thread scheduling parameters attribute

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**

<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_setschedparam()`, `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
**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().*
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**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

`pthread_attr_getscope()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_getstackaddr()**  © 2006, 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.
Classification:

POSIX 1003.1 THR TSA

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_getstackaddr()`, `pthread_attr_init()`, `pthread_create()`. 
**pthread_attr_getstacklazy()**

-get the thread lazy-stack attribute

### Synopsis:
```
#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()`
**pthread_attr_getstackprealloc()** © 2006, QNX Software Systems GmbH & Co. KG.

Get the size of memory to preallocate for a 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 value of the memory size attribute from the attribute structure `attr` to preallocate for a MAP_LAZY stack. By default, the system allocates `sysconf(_SC_PAGESIZE)` bytes of physical memory for 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 implement an 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>
</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_getstacksize()`, `pthread_attr_init()`,
`pthread_attr_setstackprealloc()`, `pthread_create()`
**pthread_attr_getstacksize()**  © 2006, QNX Software Systems GmbH & Co. KG.

*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

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

See also:

pthread_attr_getstacksize(), pthread_attr_init(), pthread_create().
Initialize a thread-attribute object

Synopsis:

```
#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:
**pthread_attr_init()**

<table>
<thead>
<tr>
<th>Get</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_getdetachstate()</td>
<td>pthread_attr_setdetachstate()</td>
</tr>
<tr>
<td>pthread_attr_getguardsize()</td>
<td>pthread_attr_setguardsize()</td>
</tr>
<tr>
<td>pthread_attr_getinheritsched()</td>
<td>pthread_attr_setinheritsched()</td>
</tr>
<tr>
<td>pthread_attr_getschedparam()</td>
<td>pthread_attr_setschedparam()</td>
</tr>
<tr>
<td>pthread_attr_getschedpolicy()</td>
<td>pthread_attr_setschedpolicy()</td>
</tr>
<tr>
<td>pthread_attr_getscope()</td>
<td>pthread_attr_setscope()</td>
</tr>
<tr>
<td>pthread_attr_getstackaddr()</td>
<td>pthread_attr_setstackaddr()</td>
</tr>
<tr>
<td>pthread_attr_getstacklazy()</td>
<td>pthread_attr_setstacklazy()</td>
</tr>
<tr>
<td>pthread_attr_getstacksize()</td>
<td>pthread_attr_setstacksize()</td>
</tr>
</tbody>
</table>

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**

<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…
### pthread_attr_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_attr_destroy()`, `pthread_create()`
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.
**Returns:**

- **EOK**  Success.
- **EINVAL**  Invalid thread detach state value.

**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_attr_getdetachstate()`, `pthread_attr_init()`, `pthread_create()`, `pthread_detach()`, `pthread_join()`.
**pthread_attr_setguardsize()**

*Set the size of the thread’s guard area*

**Synopsis:**

```c
#include <pthread.h>

int pthread_attr_setguardsize(
    pthread_attr_t* attr,
    size_t guardsize );
```

**Arguments:**

- `attr` is 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` is 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 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

<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:

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()`
Set a thread’s inherit-scheduling attribute

Synopsis:

```c
#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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EINV</td>
<td>Invalid thread attribute object <em>attr</em>.</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>Invalid thread inherit scheduling attribute <em>inheritsched</em>.</td>
</tr>
</tbody>
</table>

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_getinheritsched()`, `pthread_attr_init()`, `pthread_create()`. 
Set a thread’s scheduling parameters attribute

Synopsis:
```
#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.
- `EINVAL` Invalid thread attribute object `attr`.
- `ENOTSUP` Invalid thread scheduling parameters attribute `param`.

© 2006, QNX Software Systems GmbH & Co. KG.
pthread_attr_setschedparam()

Classification:

POSIX 1003.1 THR

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_getschedparam(), pthread_attr_setinheritsched(),
pthread_attr_init(), pthread_create(), sched_param

Set the thread scheduling policy attribute

**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_setschedpolicy()`, `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`.

© 2006, QNX Software Systems GmbH & Co. KG.
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_getscope()`, `pthread_attr_init()`, `pthread_create()`.
**pthread_attr_setstackaddr()** © 2006, QNX Software Systems GmbH & Co. KG.

*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>
</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 interpretation of PTHREAD_STACK_MIN is enough memory to run a thread that does nothing:

```c
void nothingthread( void )
{
    return;
}
```

See also:

`pthread_attr_getstackaddr()`, `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 `-l c` 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.

- **EINVAL**  
  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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_getstacklazy()`, `pthread_attr_setinheritsched()`
**pthread_attr_setstackprealloc()** © 2006, QNX Software Systems GmbH & Co.

Set the size of memory to preallocate for a stack

Synopsis:

```c
#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 stack size you want to preallocate for the new 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_setstackprealloc()` function sets the size of the memory to preallocate for a MAP_LAZY stack. By default, the system allocates `sysconf(_SC_PAGESIZE)` bytes of physical memory for 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.
pthread_attr_setstackprealloc()

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_getstackprealloc(), pthread_attr_getstacksize(),
 pthread_attr_init(), pthread_create()
**pthread_attr_setstacksize()**  © 2006, QNX Software Systems GmbH & Co. KG.

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

<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:

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()**
© 2006, QNX Software Systems GmbH & Co. KG.

*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()`
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:

```c
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

<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_barrierattr_init()`, `pthread_barrier_destroy()`, `pthread_barrier_wait()`
**pthread_barrier_wait()**

Synchronize participating threads at the barrier

**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.

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

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_barrier_init()
**pthread_barrierattr_destroy()** © 2006, QNX Software Systems GmbH & Co. KG.

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
**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 `-l c` 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`. 
`pthread_barrierattr_getpshared()`

### 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()`
**pthread_barrierattr_init()**

Initialize a barrier attributes object

**Synopsis:**

```c
#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:**

```c
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
  - `continued...`
See also:

`pthread_barrier_init()`, `pthread_barrierattr_destroy()`, `pthread_barrierattr_getpshared()`, `pthread_barrierattr_setpshared()`
Set the value of a barrier's process-shared attribute

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`. 
**pthread_barrierattr_setpshared()**

**Returns:**

- **EOK** Success.
- **EINVAL** 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_getpshared()`, `pthread_barrierattr_init()`
**pthread_cancel()**

**Cancel a thread**

**Synopsis:**

```c
#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

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_cleanup_pop(), pthread_cond_timedwait(),
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
**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()** © 2006, QNX Software Systems GmbH & Co. KG.

*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.
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()**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety</td>
<td></td>
</tr>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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()**

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()**

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

**See also:**

- `pthread_cond_signal()`, `pthread_cond_wait()`, `SyncCondvarSignal()`
Synopsis:

```c
#include <pthread.h>

int pthread_cond_destroy( pthreadCond_t* cond );
```

Arguments:

- `cond` A pointer to the `pthreadCond_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>
<tbody>
<tr>
<td>continued...</td>
<td></td>
</tr>
</tbody>
</table>
pthread_cond_destroy() © 2006, 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>Yes</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

pthread_cond_init(), SyncDestroy()
**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:**

```c
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:
The condition variable can only be accessed by threads created within the same process as the thread that initialized the condition variable.

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.
- **EAGAIN** All kernel synchronization objects are in use.
- **EBUSY** Previously initialized condition variable, `cond`, hasn’t been destroyed.
- **EFAULT** A fault occurred when the kernel tried to access `cond` or `attr`.
- **EINVAL** The value specified by `cond` 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_condattr_init()`, `pthread_cond_destroy()`
**pthread_cond_signal()** © 2006, QNX Software Systems GmbH & Co. KG.

*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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_wait()`, `SyncCondvarSignal()`.
**pthread_cond_timedwait()**

*Wait on a condition variable, with a time limit*

**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;
}

Classification:

POSIX 1003.1 THR
**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_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
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 );
    }
```
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

<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:

`pthread_cond_broadcast()`, `pthread_cond_init()`,
`pthread_cond_signal()`, `pthread_cond_timedwait()`, `SyncCondvarWait()`
**pthread_condattr_destroy()**

Destroy a condition-variable attribute object

Synopsis:
```
#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
See also:

pthread_condattr_init(), pthread_cond_init()
**pthread_condattr_getclock()** © 2006, QNX Software Systems GmbH & Co. KG.

**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:

```c
#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`.
**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_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 `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_condattr_destroy()`, `pthread_cond_init()`
`pthread_condattr_setclock()` © 2006, 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.
EINV AL   Invalid value attr.

Classification:

POSIX 1003.1 THR CS

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_init(), pthread_cond_timedwait(),
pthread_condattr_destroy(), pthread_condattr_getclock(),
pthread_condattr_getpshared(), pthread_condattr_init(),
pthread_condattr_setpshared(), pthread_create()
**pthread_condattr_setpshared()** © 2006, QNX Software Systems GmbH & Co.

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 `qcc` 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>
</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_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_*` 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_CANCELASYNCHRONOUS**
  
  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

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_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**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>continued…</td>
<td></td>
</tr>
</tbody>
</table>
### 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:**

`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

**Safety**

- Cancellation point: No
  continued…
### pthread_equal()

**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_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**

<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:**

`atexit()`, `exit()`, `pthread_create()`, `pthread_cleanup_push()`, `pthread_cleanup_pop()`, `pthread_join()`, `ThreadDestroy()`.
### Synopsis:

```c
#include <pthread.h>

int pthread_getconcurrency( void );
```

### Library:

**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()`
pthread_getcpuclockid()  
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_getcpu_clockid()**

© 2006, 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_gettime(), clock_getres(), clock_gettimeofday(), ClockId(), clock_settime(), pthread_getcpu_clockid(), 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.

Returns:

- **EOK**: Success.
- **E2BIG**: The name is larger than the accepted size.
**pthread_getname_np()**

```
EINVAL    The name buffer length is invalid or smaller that the new name length.
EPERM    The process doesn’t have superuser capabilities.
```

**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_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`.

November 2, 2006
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_create()`, `pthread_setschedparam()`, `sched_param`
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:**

```c
#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()`, `ThreadJoinr()`
## pthread_key_create()

Create a thread-specific data key

**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() );
    }
    return 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**

<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>
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()`
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()**

© 2006, 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_key_create()`
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:

- `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`.
pthread_kill()

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(), ThreadDestroy()
pthread_mutex_destroy()
Destroy a mutex

Synopsis:
#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*

**Synopsis:**
```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_setprioceiling()
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_RMW_MUTEX_INITIALIZER` (for recursive mutexes).
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:**

```
#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:**

`libc`

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.

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_setrecursion() 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++;
    }
}
```

```c
void* function2( void* arg )
{
    int tmp = 0;

    while( 1 )
    {
        pthread_mutex_lock( &mutex );
        tmp = count++;
    }
}
```

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

Safety

Cancellation point  No

continued…
**pthread_mutex_lock()**

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_mutex_init()`, `pthread_mutexattr_setrecursiv()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, `SyncMutexLock()`
Synopsis:

```
#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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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()`
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.

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 \textit{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 \texttt{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.

\textbf{Returns:}

Zero on success, or an error number to indicate the error.

\textbf{Errors:}

- \texttt{EAGAIN} The mutex couldn’t be acquired because the maximum number of recursive locks for the mutex has been exceeded.
- \texttt{EDEADLK} The current thread already owns the mutex.
- \texttt{EINVAL} The mutex was created with the protocol attribute having the value \texttt{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 \textit{abs\_timeout} parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million; or the value specified by \texttt{mutex} doesn’t refer to an initialized mutex object.
- \texttt{ETIMEDOUT} The mutex couldn’t be locked before the specified timeout expired.

\textbf{Classification:}

POSIX 1003.1 THR TMO
**pthread_mutex_timedlock()**

<table>
<thead>
<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_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`, `timespec`
**pthread_mutex_trylock()** © 2006, 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

<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:**

`pthread_mutex_lock()`, `pthread_mutex_unlock()`
pthread_mutex_unlock() © 2006, 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 `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

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

`pthread_mutex_lock()`, `pthread_mutex_trylock()`, `SyncMutexUnlock()`
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
**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_mutexattr_init()`, `pthread_mutex_init()`
Get the priority ceiling of a mutex attribute object

Synopsis:

```
#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>
</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_getprotocol()`, `pthread_mutexattr_getrecursive()`, `pthread_mutexattr_getprioceiling()`, `pthread_mutexattr_setrecursive()`
Get a mutex’s scheduling protocol

Synopsis:

```c
#include <pthread.h>

int pthread_mutexattr_getprotocol(  
    pthread_mutexattr_t * 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:

```c
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_mutexattr_getprotocol()**

**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_PRIO_NONE 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>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:

`pthread_mutexattr_getprotocol()`, `pthread_mutexattr_setprotocol()`, `pthread_mutexattr_setrecursive()`
 pthread_mutexattr_getpshared()
Get the process-shared attribute from a mutex attribute object

Synopsis:
#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.
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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_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.
- `pshared` 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`. 
$\texttt{pthread_mutexattr_getrecursive()}$ © 2006, QNX Software Systems GmbH & Co. KG.

**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:**

$\texttt{pthread_mutexattr_init()}, \texttt{pthread_mutexattr_setrecursive()}$
**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.
pthread_mutexattr_gettype()  © 2006, QNX Software Systems GmbH & Co. KG.

- 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:
EINVAL  Invalid value specified by attr.

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>
</tbody>
</table>

continued...
Safety

| Thread | Yes |

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_timedwait()`
`pthread_cond_timedwait()`
`pthread_mutexattr_settype()`
**pthread_mutexattr_init()**

Initialize a mutex attribute object

**Synopsis:**

```c
#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` `THREAD_PRIO_INHERIT`
- `recursive` `THREAD_RECURSIVE_DISABLE`

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>pthread_mutexattr_getprioceiling()</td>
<td>pthread_mutexattr_setprioceiling()</td>
</tr>
<tr>
<td>pthread_mutexattr_getprotocol()</td>
<td>pthread_mutexattr_setprotocol()</td>
</tr>
<tr>
<td>pthread_mutexattr_getpshared()</td>
<td>pthread_mutexattr_setpshared()</td>
</tr>
<tr>
<td>pthread_mutexattr_getrecursion()</td>
<td>pthread_mutexattr_setrecursion()</td>
</tr>
<tr>
<td>pthread_mutexattr_gettype()</td>
<td>pthread_mutexattr_settype()</td>
</tr>
</tbody>
</table>

**Returns:**

- **EOK** Success.
- **ENOMEM** Insufficient memory to initialize mutex attribute object `attr`.

**Classification:**

- POSIX 1003.1 THR

**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_mutex_init()`, `pthread_mutexattr_destroy()`,
`pthread_mutexattr_getprioceiling()`, `pthread_mutexattr_getprotocol()`,
`pthread_mutexattr_getpshared()`, `pthread_mutexattr_getrecursion()`,
`pthread_mutexattr_gettype()`, `pthread_mutexattr_setprioceiling()`,
pthread_mutexattr_init() © 2006, QNX Software Systems GmbH & Co. KG.

pthread_mutexattr_setprotocol(), pthread_mutexattr_setpshared(),
pthread_mutexattr_setrecursive(), pthread_mutexattr_settype()
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

<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_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_t *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_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:

POSIX 1003.1 THR TPP | TPI

<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_getprotocol()`, `pthread_mutexattr_getrecursive()`
**Synopsis:**

```c
#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:**

- **EOK**  
  Success.

- **EINVAL**
  Invalid mutex attribute object, `attr`.

  The new value specified in `pshared` isn’t `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

**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_mutexattr_getpshared()`, `pthread_mutexattr_getrecursive()`,  
`pthread_mutex_init()`, `pthread_mutexattr_setrecursive()`
**pthread_mutexattr_setrecursive()**

*Set the recursive attribute in mutex attribute object*

**Synopsis:**

```c
#include <pthread.h>

int pthread_mutexattr_setrecursive(
   pthread_mutexattr_t* attr,
   int recursive );
```

**Arguments:**

- `attr`  
  A pointer to the `pthread_mutexattr_t` object that you want to set the attribute in.

- `recursive`  
  The new value for the recursive attribute; one of:

  - PTHREAD_RECURSIVE_ENABLE — a thread that has already locked the mutex can lock it again without blocking.
  
  - PTHREAD_RECURSIVE_DISABLE — any thread that tries to lock the mutex blocks, if that mutex is already locked.

  The default value of the recursive attribute is PTHREAD_RECURSIVE_DISABLE.

**Library:**

- libc

  Use the `-l c` 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`. 
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>
</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_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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</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_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 `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

<table>
<thead>
<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>

pthread_once()
**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 c` 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>
</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_init()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_tryrdlock()`, `pthread_rwlock_trywrlock()`, `pthread_rwlock_unlock()`, `pthread_rwlock_wrlock()`
**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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</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:

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()**

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*.
EINVAL    The read-write lock *rwl* is invalid.

**Classification:**

POSIX 1003.1 THR

**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:**

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**

<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...
**pthread_rwlock_timedrdlock()** © 2006, 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:

- `pthread_rwlock_destroy()`, `pthread_rwlock_init()`,
- `pthread_rwlock_timedwrlock()`, `pthread_rwlock_trywrlock()`,
- `pthread_rwlock_ryrdlock()`, `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**  
  Couldn’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_rdlock(), pthread_rwlock_unlock(),
pthread_rwlock_wrlock(), timespec
**pthread_rwlock_tryrdlock()** © 2006, QNX Software Systems GmbH & Co. KG.
Attempt to acquire a shared lock on a read-write lock

**Synopsis:**

```
#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`. 
**pthread_rwlock_tryrdlock()**

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()  © 2006, QNX Software Systems GmbH & Co. KG.
Attempt to acquire an exclusive lock on a read-write lock

Synopsis:

```
#include <pthread.h>

int pthread_rwlock_trywrlock(
    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_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.
pthread_rwlock_trywrlock()

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>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:

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`.

© 2006, QNX Software Systems GmbH & Co. KG.
EINV AL 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()**

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`. 

© 2006, QNX Software Systems GmbH & Co. KG.
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>
<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_trywrlock(), pthread_rwlock_unlock()
**pthread_rwlockattr_destroy()** © 2006, QNX Software Systems GmbH & Co. KG.

Destroy a read-write lock attribute object

**Synopsis:**

```c
#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
**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_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:

```c
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 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()`
**pthread_rwlockattr_init()**

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

Synopsis:

```
#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 libc` 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

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:

pthread_rwlockattr_destroy(), pthread_rwlockattr_getpshared(), pthread_rwlockattr_init()
**Synopsis:**

```c
#include <pthread.h>

pthread_t pthread_self( void );
```

**Library:**

`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**.
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:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EOK</td>
<td>Success.</td>
</tr>
<tr>
<td>EINV AL</td>
<td>The cancellation state specified by <code>state</code> is invalid.</td>
</tr>
</tbody>
</table>

**Classification:**

POSIX 1003.1 THR

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

**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()`
pthread_setconcurrency()

Set the concurrency level for a thread

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

<table>
<thead>
<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_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 libc` 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 that the new name length.
- `EPERM` The process doesn’t have superuser capabilities.
**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_getname_np()`
Synopsis:

#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 × 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`.
** pthread_setspecific()  
© 2006, QNX Software Systems GmbH & Co. KG.

** 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:**

```c
#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
pthread_sigmask() © 2006, QNX Software Systems GmbH & Co. KG.

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.

Returns:

EOK Success.

EINVAL Invalid how parameter.

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:

sigprocmask()
**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...
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>EINV AL</td>
<td>Invalid sleepon address.</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:**

`pthread_cond_broadcast()`, `pthread_sleepon_signal()`, `pthread_sleepon_lock()`, `pthread_sleepon_unlock()`, `pthread_sleepon_wait()`
**Synopsis:**
```
#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
```
**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_mutex_lock()`, `pthread_sleepon_broadcast()`, `pthread_sleepon_signal()`, `pthread_sleepon_unlock()`, `pthread_sleepon_wait()`
**Synopsis:**

```c
#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

<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_sleepon_broadcast()`, `pthread_sleepon_lock()`, `pthread_sleepon_unlock()`, `pthread_sleepon_wait()`
#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:

pthread_sleepon_lock();
while((ptr = cachelist->free) == NULL) {
    pthread_sleepon_timedwait(cachelist, 1000);
}  

cachelist->free = ptr->free;

pthread_sleepon_unlock();

To start an operation and wait upon its completion:

/* 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 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_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();
    }
}
```

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_timedwait() © 2006, QNX Software Systems GmbH & Co. KG.

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>
</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_unlock()`, `pthread_sleepon_broadcast()`,
- `pthread_sleepon_lock()`, `pthread_sleepon_signal()`,
- `pthread_sleepon_wait()`
**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);
}
ptr->free = cachelist->free;
```

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 `addr`
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.

Returns:

- **EOK**  : Success.
- **EDEADLK**  : The calling thread already owns the controlling mutex.

Classification:

- **QNX Neutrino**

<table>
<thead>
<tr>
<th>Safety</th>
<th>Yes</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()`
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 `-l 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

**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_lock(), pthread_spin_trylock(),
pthread_spin_unlock()
**pthread_spin_lock()**

Lock a thread spinlock

**Synopsis:**

```
#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`.
Classification:

POSIX 1003.1 THR SPI

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:

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:

`pthread_spin_destroy()`, `pthread_spin_init()`, `pthread_spin_trylock()`, `pthread_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`. 

© 2006, QNX Software Systems GmbH & Co. KG.
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_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
### 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_spin_destroy()`, `pthread_spin_init()`, `pthread_spin_lock()`, `pthread_spin_trylock()`
**pthread_testcancel()**

*Test thread cancellation*

**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()`
**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>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...
**pthread_timedjoin()**

See also:

- `pthread_create()`, `pthread_detach()`, `pthread_exit()`, `pthread_join()`, `ThreadJoin()`, `ThreadJoin_r()`, `timespec`
Structure that describes a pulse

Synopsis:

```c
#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_MINAVAIL` through `_PULSE_CODE_MAXAVAIL`. 
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
**pulse_attach()**

Attach a handler function to a pulse code

**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>`:
  ```
  #ifdef MSG_FLAG_ALLOC_PULSE
  #define MSG_FLAG_ALLOC_PULSE 1
  #endif
  ```
  - `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_MINAVAIL` and `_PULSE_CODE_MAXAVAIL`.
- **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`. 
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 \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()

<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(), pulse_detach()

“Components of a Resource Manager” section of the Writing a Resource Manager chapter in the Programmer’s Guide.
Detach a handler function from a pulse code

Synopsis:

```c
#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:

```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;
    }
    ...
    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
**pulse_detach()**

© 2006, 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>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:

```c
#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:

```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 ) {
```
putc()

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:

errno, ferror(), fopen(), fputc(), fputchar(), fputs(), getc(), getc_unlocked(), getchar(), getchar_unlocked(), putchar(), putchar_unlocked(), putc_unlocked(), puts()
Synopsis:

```c
#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

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>
</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>No</td>
</tr>
</tbody>
</table>

See also:

getc(), getchar(), getchar_unlocked(), getc_unlocked(), putc(),
putchar(), putchar_unlocked()
Synopsis:

#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:

fputc( c, stdout );

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" );
    c = fgetc( fp );
    while( c != EOF ) {


November 2, 2006  C Library — P to R  2423
```c
putchar( c );
c = fgetc( fp );
}
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, fputc(), fputchar(), fputs(), getc(), getc_unlocked(), getchar(),
guchar_unlocked(), putc(), putchar_unlocked(), putc_unlocked()`
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()
putenv()

Add, change or delete an environment variable

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.

```c
#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:
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()`
puts()  Write a string to stdout

Synopsis:

```
#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:

```
#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;
}

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:

- `char *sp_namp; /* name */`
- `char *sp_pwdp; /* encrypted password */`
- `long sp_lstchg; /* 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 */`
putspent()

If the `sp_min`, `sp_max`, `sp_lstchg`, `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()`
- `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         pdbuf[80], ndbuf[80];

  memset(&sp, 0, sizeof(sp));
  if (argc < 2) {
    printf("hs filename \n", argv[0]);

```
```c

return(EXIT_FAILURE);
}

if (!(fp = fopen(argv[1], "w"))) {
    fprintf(stderr, "Can't open file %s 
", argv[1]);
    perror("Problem ");
    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>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:

`errno`, `getgrent()`, `getlogin()`, `getpwnam()`, `getpwuid()`, `getspent()`, `getspnam()`, `setspent()`
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.

Note: 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 `USERPROCESS` or `DEADPROCESS`; 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

<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 machine-dependent, and might not be
read correctly using getw() on a different processor.

See also:

errno, ferror(), fopen(), fputc(), fputchar(), fputs(), getw(), putchar(),
putchar_unlocked(), puc_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).

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()**

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, getwc(), getwchar()`

“Stream I/O functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```c
#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 `/dev/stdout` stream. It’s equivalent to:

```c
fputwc( wc, stdout );
```

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.
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

<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, 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()`.
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.
- **EFBIG**: 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

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
</table>

*continued...*
### pwrite(), pwrite64()

**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()`,
`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.

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:
```c
#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

<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 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`
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.
qsort()

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

<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:

`bsearch()`
Raccept()
Accept a connection on a socket (via a SOCKS server)

Synopsis:
#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:

```
#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:

```
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%*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 {
```
raise()

```c
printf( "\nUser raised signal \%d.\n",  
        signal_number );
}

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:

signal(), SignalAction()
**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

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>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**

<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:

rand(), srand()
random()  © 2006, QNX Software Systems GmbH & Co. KG.
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 $16 \times (231-1)$. 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().
difference is that \( \text{rand}() \) produces a much less random sequence. In fact, the low dozen bits generated by \( \text{rand}() \) go through a cyclic pattern. All the bits generated by \( \text{random}() \) are usable. For example,

\[
\text{random()} \& 01
\]

produces a random binary value.

Unlike \( \text{srand}() \), \( \text{srandom}() \) doesn’t return the old seed because the amount of state information used is much more than a single word. The \( \text{initstate}() \) and \( \text{setstate}() \) 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 \( \text{rand}() \), \( \text{random}() \) produces by default a sequence of numbers that can be duplicated by calling \( \text{srandom}() \) with 1 as the seed.

If \( \text{initstate}() \) hasn’t been called, \( \text{random}() \) behaves as though \( \text{initstate}() \) had been called with \( \text{seed}=1 \) and \( \text{size}=128 \).

If \( \text{initstate}() \) is called with \( \text{size} \) less than 8, \( \text{random}() \) uses a simple linear congruential random number generator.

**Returns:**

The generated pseudo-random number.

**Examples:**

See \( \text{initstate}() \).

**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>
</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>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:

- `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).
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:

`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 gcc 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.

**If `fd2p` is:**  **Then:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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 <code>*fd2p</code>. 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

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
**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).
Rconnect()

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:

connect() Raccept(), Rbind(), Rgetsockname(), Rlisten(), Rcmd(), Rselect(), SOCKSinit()

SOCKS — A Basic Firewall
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

<table>
<thead>
<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:

tcischars()
re_comp()

Compile a regular expression

Synopsis:

```c
#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 \textit{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

\begin{table}[h]
\centering
\begin{tabular}{|l|c|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}
\end{table}

See also:

\textit{re_exec()}, \textit{regcomp()}, \textit{regerror()}, \textit{regexec()}

\texttt{grep} in the \textit{Utilities Reference}
re_exec()

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.
Reexec():

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.
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:**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed in the read operation.</td>
</tr>
<tr>
<td>EBADF</td>
<td>The file descriptor, <code>fildes</code>, isn’t a valid file descriptor open for reading.</td>
</tr>
<tr>
<td>EINTR</td>
<td>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.</td>
</tr>
<tr>
<td>EIO</td>
<td>A physical I/O error occurred (for example, a bad block on a disk). The precise meaning is device-dependent.</td>
</tr>
</tbody>
</table>
The `read()` function isn’t implemented for the filesystem specified by `fd`. The file is a regular file and an attempt is made to read at or beyond the offset maximum associated with the file.

**Examples:**
```
#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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Yes/No</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()`, `creat()`, `dup()`, `dup2()`, `errno`, `fcntl()`, `lseek()`, `open()`, `pipe()`, `readblock()`, `readv()`, `select()`, `write()`, `writev()`
Synopsis:

```c
#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_config_fname;
    char* view_config_fname;
    char* context_config_fname;
    char* acl_config_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_config_fname
  
  *party.conf* file location.

* view_config_fname
  
  *view.conf* file location.

* context_config_fname
  
  *context.conf* file location.

* acl_config_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.

trap_sink

    Destination host to send trap messages.

trap_community

    Community name to use for trap messages.

conf_authentraps

    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
read_main_config_file()

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:

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
readblock()
Read blocks of data from a file

Synopsis:

#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 gcc 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 \texttt{read()} function’s limit of INT\_MAX bytes at a time.)

If \texttt{numblks} is zero, \texttt{readblock()} returns zero and has no other results.

On successful completion, \texttt{readblock()} returns the number of blocks actually read and placed in the buffer. This number is never greater than \texttt{numblks}. The value returned may be less than \texttt{numblks} if one of the following occurs:

- The number of blocks left before the end-of-file is less than \texttt{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, \texttt{readblock()} returns -1 and sets \texttt{errno} to EIO.

\textbf{Returns:}

The number of blocks actually read. If an error occurs, it returns -1, sets \texttt{errno} to indicate the error, and the contents of the buffer pointer to by \texttt{buff} are left unchanged.

\textbf{Errors:}

- \texttt{EBADF} The \texttt{fd} argument isn’t a valid file descriptor open for reading a block-oriented device.

- \texttt{EIO} A physical read error occurred on the first block.

- \texttt{EINVAL} The starting position is invalid (0 or negative) or beyond the end of the disk.

\textbf{Classification:}

QNX Neutrino
**readblock()**

### 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:**

`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 (min, time, and timeout), when used on terminal devices in RAW mode, override the behavior of the MIN and TIME elements of the currently defined termios structure (for the duration of this call only). The termios structure also defines a forwarding character (in \texttt{c.cc[VFWD]}) that can be used to bypass min, time and timeout.

The normal case of a simple read by an application would block until at least one character was available.

![Diagram](image)

\begin{center}
\begin{tabular}{l l}
\text{MIN} & \text{TIME} & \text{TIMEOUT} & \text{FORWARD} \\
\text{Read} & \text{when at least} & \text{Respond if a pause} & \text{Respond if a framing} \\
\text{n characters} & \text{this number} & \text{in the character} & \text{character arrives.} \\
\text{arrives.} & \text{of characters} & \text{stream occurs.} & \\
\text{MIN} & \text{TIME} & \text{READ} & \text{FORWARD} \\
\text{Read} & \text{n bytes} & \text{Char I/O} & \text{DLL} \\
\end{tabular}
\end{center}

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.

**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.

**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 f wd _char = 0x7 e;

#include <termios.h>
```

```c
int fd;
struct termios termio;

:
:
tcgetattr(fd, &termio);
termio.c_c[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 <code>n</code> bytes).</td>
</tr>
<tr>
<td><code>M</code></td>
<td>0</td>
<td>0</td>
<td>Return with up to <code>n</code> bytes only when at least <code>M</code> bytes are available.</td>
</tr>
<tr>
<td>0</td>
<td><code>T</code></td>
<td>0</td>
<td>Return with up to <code>n</code> bytes when at least one byte is available, or <code>T * .1 sec</code> has expired.</td>
</tr>
<tr>
<td><code>M</code></td>
<td><code>T</code></td>
<td>0</td>
<td>Return with up to <code>n</code> bytes when either <code>M</code> bytes are available, or at least one byte has been received and the inter-byte time between any subsequently received characters exceeds <code>T * .1 sec</code>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td><code>t</code></td>
<td>RESERVED.</td>
</tr>
<tr>
<td><code>M</code></td>
<td>0</td>
<td><code>t</code></td>
<td>Return with up to <code>n</code> bytes when <code>t * .1 sec</code> has expired, or <code>M</code> bytes are available.</td>
</tr>
<tr>
<td>0</td>
<td><code>T</code></td>
<td><code>t</code></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 ) .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 ) .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 `MIN` and `TIME` parameters of the `termios` controlling structure. Thus, `readcond()` can be used as a higher speed alternative to consecutive calls of `tcgetattr()`, `tcsetattr()`, and `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 `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 (`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 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`. 

---

© 2006, QNX Software Systems GmbH & Co. KG.
ENOSYS  This function isn’t supported for this \textit{fd}.

**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:**

\texttt{errno, read(), tcgetattr(), tcsetattr(), termios}
**Synopsis:**

```c
#include <dirent.h>

struct dirent * readdir( DIR * dirp );
```

**Arguments:**

- `dirp` A pointer to the directory stream to be read.

**Library:**

- `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:

.. code-block::

   .DEXTRA_FIRST( pdirent )
   
   Get a pointer to the first block of data associated with the 
   structure pointed to by \textit{pdirent}.

.. _DEXTRA_NEXT:

.. code-block::

   .DEXTRA_NEXT( last)
   
   Get the block of data that follows the block pointed to by \textit{last}.

.. _DEXTRA_VALID:

.. code-block::

   .DEXTRA_VALID( extra, pdirent)
   
   Evaluates to 1 if \textit{extra} is a pointer to a valid block of data 
   associated with the structure pointed to by \textit{pdirent}.

You can use these macros to traverse the data associated with the 
\textit{dirent} structure like this:

.. code-block::

   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 \texttt{struct dirent} object for success, or NULL if the end of the directory stream is encountered or an error occurs (\textit{errno} is set).

**Errors:**

EBADF The \textit{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
", direntp->d_name );
        }
        closedir( dirp );
        return EXIT_SUCCESS;
    }
    return EXIT_FAILURE;
}
```

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>
</tbody>
</table>

continued…
readdir()

Safety

<table>
<thead>
<tr>
<th>Signal handler</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

closedir(), errno, lstat(), opendir(), readdir_r(), rewinddir(), seekdir(), telldir(), stat()
readdir_r()

Get information about the next matching filename

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()`
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:

- 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

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

**See also:**

`errno`, `lstat()`, `symlink()`
readv()

Synopsis:

```
#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, \texttt{readv()} returns -1 and sets \texttt{errno} to EAGAIN.

If a process has the pipe open for writing, and O_NONBLOCK is clear, \texttt{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, \texttt{readv()} returns -1 and sets \texttt{errno} to EAGAIN.

2. If O_NONBLOCK is clear, \texttt{readv()} blocks until some data is available.

3. The O_NONBLOCK flag has no effect if some data is available.

If you call \texttt{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 \texttt{readv()} succeeds, the \texttt{st_atime} field of the file is marked for update.

\textbf{Returns:}

The number of bytes read, or -1 if an error occurred (\texttt{errno} is set).

\textbf{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, \texttt{fildes}, isn’t a valid file descriptor open for reading.

- **EINVAL**: 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.
EINV AL The iovecnt 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()
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 )
{
```

November 2, 2006
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>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:

calloc(), free(), malloc(), sbrk()
Synopsis:

```c
#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.
realpath()

Errors:
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

<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:
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>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:
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
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
recvfrom()

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

<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>
recvfrom()

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-ttcpip.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:

```
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:

```
struct cmsghdr {
```
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
recvmsg()

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:

recv(), recvfrom(), sendmsg()
regcomp()  

Compile a regular expression

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`
- cntrl
- digit
- graph
- lower
- print
- punct
- space
- upper
- xdigit

[[=c=]] All character in the equivalence class with c.
[[=.=]] All collating elements.
\{m,n\} m through n occurrences of x.
\c Character c, even if c is an operator.
\(x\) A labeled subexpression, x.
\m The mth subexpression encountered.
xy Expression x followed by y.

**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…
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>

Contributing author:

Henry Spencer. For copyright information, see Third-Party Copyright Notices in this reference.

See also:

regerror(), regexec(), regfree()
**Synopsis:**

```c
#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.
regerror()

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:

```
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 regexec() function compares string against the compiled regular expression preg. If regexec() finds a match it returns zero; otherwise, it returns nonzero.

The 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 regcomp().

The regexec() function records the matches in the pmatch array, with nmatch specifying the maximum number of matches to record. The regmatch_t structure is defined as:

```c
typedef struct {
    regoff_t rm_sp;
    regoff_t rm_ep;
} regmatch_t;
```

The members are:

- `rm_sp` The byte offset from the beginning of the string to the beginning of the matched substring.

- `rm_ep` One greater than the offset from the beginning of the string to the end of the matched substring.

The offsets in pmatch[0] identify the substring corresponding to the entire expression, while those in pmatch[1...nmatch] identify up to the first nmatch subexpressions. Unused elements of the 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

  `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>

Contributing author:

Henry Spencer. For copyright information, see Third-Party Copyright Notices in this reference.

See also:

`regcomp()`, `regerror()`, `regexec()`
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 \( |n - x/y| = 0.5 \), 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 `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:

`drem()`, `modf()`
**Synopsis:**

```c
#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()
Rename a file

Synopsis:

#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 ).
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()

EROFS  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:

```
#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

<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
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.
res_init()

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>
</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:

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
**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.
**Classification:**

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>

**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_compi() 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

<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()*,*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 `qcc` 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
<res_search()>

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_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:**

```c
#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 `qcc` 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

**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:

- `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:

```c
#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_iofuncs_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.
resmgr_attach()

- `.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.

**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><em>file_type</em>:</th>
<th>Matches pathname of type:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mq_open()</code></td>
<td>_FTYPE_MQUEUE</td>
<td><code>_FTYPE_ANY</code> _FTYPE_MQUEUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>open()</code></td>
<td>_FTYPE_ANY</td>
<td>All types</td>
</tr>
<tr>
<td><code>pipe()</code></td>
<td>_FTYPE_PIPE</td>
<td>_FTYPE_ANY or _FTYPE_PIPE</td>
</tr>
<tr>
<td><code>sem_open()</code></td>
<td>_FTYPE_SEM</td>
<td>_FTYPE_ANY or _FTYPE_SEM</td>
</tr>
<tr>
<td><code>shm_open()</code></td>
<td>_FTYPE_SHMEM</td>
<td>_FTYPE_ANY or _FTYPE_SHMEM</td>
</tr>
</tbody>
</table>

*continued…*
Function: file type: Matches pathname of type:
socket() _FTYPE_SOCKET _FTYPE_ANY or _FTYPE_SOCKET

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:

typedef struct _resmgr_attr {
    unsigned flags;
    unsigned nparts_max;
    unsigned msg_max_size;
    int (*other_func)
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.

- **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 \texttt{open()} with the matching prefix stripped off. Without this flag, the pathname is treated as a simple file requiring an exact match.

<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 &quot;&quot;</td>
<td>Match &quot;&quot;</td>
</tr>
<tr>
<td>/a/b</td>
<td>/a/b/c</td>
<td>Match \texttt{c}</td>
<td>No match</td>
</tr>
<tr>
<td>/a/b</td>
<td>/a/b/c/d</td>
<td>Match \texttt{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 \texttt{NULL}. 

November 2, 2006
resmgr_attach()

_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_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, "%s: Unable to allocate \n dispatch handle.\n", argv[0]);
    return EXIT_FAILURE;
}

/* initialize resource manager attributes */
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS, &io_funcs);

/* initialize attribute structure */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

/* attach our device name (passing in the POSIX defaults 
 from the iofunc_func_init and iofunc_attr_init functions) */
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;
}

/* allocate a context structure */
ctp = resmgr_context_alloc(dpp);

/* start the resource manager message loop */
while (1) {
    if ( (ctp = resmgr_block(ctp)) == NULL ) {
        fprintf(stderr, "block error\n");
        return EXIT_FAILURE;
    }
    resmgr_handler(ctp);
}

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

<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 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()

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:

- **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 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()*.
resmgr_block()  

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:

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.
### resmgr_connect_funcs_t

**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 _IO_CONNECT 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()  © 2006, QNX Software Systems GmbH & Co. KG.

Allocate a resource-manager context

Synopsis:

```c
#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`.
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;
    }
}
```

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>
resmgr_context_alloc()

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
**Synopsis:**

```c
#include <sys/iofunc.h>
#include <sys/dispatch.h>

void resmgr_context_free( resmgr_context_t * ctp );
```

**Arguments:**

cptr 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, "Unable to allocate \n dispatch handle.\n", argv[0] );
        return EXIT_FAILURE;
    }
}
resmgr_context_free()

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

| Cancellation point | Yes
| Interrupt handler  | No
| Signal handler     | No
| Thread             | Yes

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.
resmgr_context_t

`msg_max_size`  The minimum amount of space reserved for receiving a message.

`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()
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 `gcc` 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:

0     Success.

-1    An error occurred (errno is set).
resmgr detach()

 2006, QNX Software Systems GmbH & Co. KG.

Errors:
EINVAL

The id was never attached with resmgr attach().

ENOENT

A previous detachment request is in progress, or the id
has already been detached.

Examples:
#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 \
dispatch handle.\n",argv[0] );
return EXIT FAILURE;
}
id = resmgr attach ( ... );
.
.
.
if ( resmgr detach( dpp, id, 0) == -1 ) {
fprintf( stderr, "Failed to remove pathname \
from the pathname space.\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
Cancellation point

Yes

continued. . .

November 2, 2006

C Library — P to R

2589


resmgr_detach()

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:

dispatch_create(), resmgr_attach()

“Writing a Resource Manager” chapter of the Programmer’s Guide.
Synopsis:

```c
#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 an `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()`.
resmgr_devino()

Returns:
-1 on error (errno is set); any other value on success.

Errors:
EINVAL   The id argument is invalid.

Examples:
#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

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_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**

- Cancellation point: No
- Interrupt handler: No

*continued...*
<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>
resmgr_handle_tune()

Tune aspects of client fd-to-OCB mapping

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:

  \[
  \text{ceil}(\frac{\text{max fd}}{\text{max_client_handles}})
  \]
If this value changes, the new value takes effect for newly connected clients. Existing clients are unaffected.

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
resmgr_handle_tune()

### 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:**

*resmgr_open_bind()*

Writing a Resource Manager chapter of the *Programmer’s Guide*
**resmgr_handler()**

Handle resource manager messages

**Synopsis:**
```c
#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.

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, "%s: 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>

© 2006, QNX Software Systems GmbH & Co. KG.
Caveats:

If you attach messages or pulses to `dpp` by calling `message_attach()`, `pulse_attach()`, or `select_attach()`, 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:

```c
#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
Safety
Cancellation point No
Interrupt handler No
Signal handler No
Thread Yes

See also:
resmgr_io_funcs_t, resmgr_iofuncs()
resmgr_io_funcs_t

Table of POSIX-level I/O functions

Synopsis:

#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_OCB_T *ocb);
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);
} resmgr_io_funcs_t;

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_FDSINFO 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

<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(),
Writing a Resource Manager chapter of the QNX Neutrino
Programmer’s Guide
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.
Errors:

ESRCH   The connection can’t be located in the resource manager’s table.
ENOMEM  There is no memory available for the operation.
EINVAL  Invalid arguments were 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:

_msg_info, MsgInfo(), resmgr_io_funcs_t, _resmgr_ocb(), resmgr_open_bind(), resmgr_unbind()
**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*.

**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

<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:**

`MsgRead()`, `resmgr_context_t`, `resmgr_msgreadv()`, `resmgr_msgwrite()`, `resmgr_msgwritex()`

“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 revid 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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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.

- **ESRVFAULT**  
  A fault occurred when the kernel tried to access the buffers provided.

**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:**

`MsgReadv()`, `resmgr_context_t`, `resmgr_msgreadv()`, `resmgr_msgwritev()`

“Combine messages” in the Writing a Resource Manager chapter of the *Programmer’s Guide*
**Synopsis:**

```c
#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 c` 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.
- ESRVRFault: A fault occurred when the kernel tried to access the buffers provided.

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:

`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 parts,
    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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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

<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:**

- `MsgWritev()`, `resmgr_context_t`, `resmgr_msgread()`, `resmgr_msgreadv()`, `resmgr_msgwrite()`

“Combine messages” in the Writing a Resource Manager chapter of the *Programmer’s Guide*
Synopsis:

#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>

November 2, 2006
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()`
Retrieve an Open Control Block

Synopsis:

```
#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`.
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

<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:

iobuf_ocb_t, msg_info, MsgInfo(), resmgr_iobufcs(), resmgr_open_bind(), resmgr_unbind()
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`. 

November 2, 2006 C Library — P to R 2623
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_*). 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).
Errors:

EINV AL The id and/or info members of ctp aren’t valid.
ENOMEM Insufficient memory to allocate an internal data structure.

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.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
**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 path name.
- `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.
resmgr_pathname()

By default, this function calls:

```c
netmgr_ndtostr(ND2S_DIR_SHOW, nd, buf, sizeofbuf)
```

If you specify _RESMGR_PATHNAME_LOCALPATH, it calls

```c
netmgr_ndtostr(ND2S_DIR_SHOW|ND2S_LOCALSTR, 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

<table>
<thead>
<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>

November 2, 2006  C Library — P to R  2627
resmgr_pathname()

See also:

netmgr_ndtostr(), resmgr_attach()
Synopsis:

```
#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:

```
SETIOV (ctp->iov, msg, nbytes)
```

returning `_RESMGR_NPARTS` (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:

resmgr_context_t, RESMGR_NPARTS(), RESMGR_STATUS(), SETIOV()
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

*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>

See also:

resmgr_connect_funcs_t, resmgr_io_funcs_t,
_RESMGR_NPARTS(), _RESMGR_PTR()
resmgr_unbind()
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:

- `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.
resmgr_unbind()

Errors:

ESRCH The binding can’t be located in the resource manager 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>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:

```c
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;
    }
```
```c
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**

<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:

`clearerr()`, `fopen()`, `fseek()`
rewinddir() — Rewind a directory stream to the start of the directory

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>
```
```c
#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

*continued...*
### Safety

<table>
<thead>
<tr>
<th></th>
<th>Safety</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:**

`closedir()`, `opendir()`, `readdir()`, `readdir_r()`, `seekdir()`
Synopsis:

#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()**

© 2006, QNX Software Systems GmbH & Co. KG.

### 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:

getsockname(), Raccept(), Rbind(), Rconnect(), Rlisten(), Rcmd(), Rselect(), SOCKSinit()

SOCKS — A Basic Firewall
Find the last occurrence of a character in a string

Synopsis:

```c
#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()  © 2006, QNX Software Systems GmbH & Co. KG.

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:

\textit{index()}, \textit{strchr()}, \textit{strrchr()}
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 <code>x</code> is:</th>
<th><code>rint()</code> returns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>±Infinity</td>
<td><code>x</code></td>
</tr>
<tr>
<td>NAN</td>
<td>NAN</td>
</tr>
</tbody>
</table>

November 2, 2006  C Library — P to R  2645
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**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes
See also:

ceil(), floor()
Listen for connections on a socket (via a SOCKS server)

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()

<table>
<thead>
<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:

listen() Raccept(), Rbind(), Rconnect(), Rgetsockname(), Rcmd(), Rselect(), SOCKSinit()

SOCKS — A Basic Firewall
Synopsis:
```
#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_ctime` and `st_mtime` for update in the parent directory.
rmdir()

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>
</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:

chdir(), chmod(), errno, getcwd(), mkdir(), stat()
Synopsis:

```c
#include <sys/socket.h>
#include <net/if.h>
#include <net/route.h>

int socket( PFROUTE,
            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 `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>ENOBUFFS</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 `rtm_errno` are available through the normal `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 `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 `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 `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_type; /* future binary compatibility */
  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_type; /* message type */
  int ifm_addrs; /* like rtm_addrs */
  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_type; /* future binary compatibility */
  u_char ifam_version; /* message type */
  int ifam_addrs; /* like rtm_addrs */
  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:
ROUTE

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_sthresh; /* 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_initx 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_RECVPIPE 0x8 /* init or lock _recvpipe */
#define RTV_SENDPIPE 0x10 /* init or lock _sendpipe */
#define RTV_SSTHRESH 0x20 /* init or lock _sthresh */
#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 */

November 2, 2006 C Library — P to R 2657
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>\n",
            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;
mask = &my_rt.mask;

rtm->rtm_type = RTM_ADD;
rtm->rtm_flags = RTF_UP | RTF_GATEWAY | RTF_STATIC;
rtm->rtm_mslen = sizeof(my_rtm);
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_mslen) < 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-tcpip.so in the Utilities Reference
**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>
</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:

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
**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 (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.

- **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.

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
Reserve a system resource for a process

Synopsis:

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_attach( rsr_request_t * list,
                     int count );
```

Arguments:

- `list` An array of `rsr_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:
rsrdbmgr_attach()

- rsrdbmgr_attach()
- rsrdbmgr_detach()

The rsrdbmgr_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 rsrdbmgr_detach() call). The requested resources
are returned in a list of _rsr_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
rsrdbmgr_detach() with the same start, end, and type (via the flags
field) that were issued for the resource.

rsr_request_t structure

The resource requests structure looks like this:

```c
typedef struct _rsr_request {
    uint64_t length;
    uint64_t align;
    uint64_t start;
    uint64_t end;
    uint32_t flags;
    uint32_t zero[3]; /* Reserved */
} rsr_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;
    rsrcrequest_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""%lx 0x""%lx inclusive. 
",
            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 ");
        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.
 */
```
```c
#include <stdio.h>
#include <sys/rsrcreq.h>
#include <sys/rsrcreqmsg.h>

int main(int argc, char **argv) {
    int count;
    rsrcreq_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_FLAGS_RANGE | RSRCDBMGR_FLAGS_PORT;
    req[1].start = 10;
    req[1].end = 12;
    req[1].length = 1;
    req[1].flags = RSRCDBMGR_FLAGS_RANGE | RSRCDBMGR_FLAGS_IRQ;
    count = 2;
    if (rsrcreq_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 (rsrcreq_detach(req, count) == -1) {
        perror("Problem detaching resource \n");
        exit(1);
    }
    return(0);
}
```
**rsrcregmgr_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:**

rsrcregmgr_create(), rsrcregmgr_detach(), rsrcregmgr_destroy(),
rsrcregmgr_devno_attach(), rsrcregmgr_devno_detach(),
rsrcregmgr_query()
Synopsis:

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_create( rsr_alloc_t *item,
                     int count );
```

Arguments:

- `item` An array of `rsr_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 `rsrdbmgr_create()` function creates one or more system resources. If the function completes successfully, `count` resources are returned in `item`.

`rsr_alloc_t` structure

The structure of a basic resource request looks like this:

```c
typedef struct _rsr_alloc {
    uint64_t start; /* Start of resource range */
    uint64_t end;  /* End of resource range */
    uint32_t flags; /* Resource type | Resource flags */
} rsr_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/rsrcdbmgr.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/rsrcdbmgr.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 \n");
        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>
</table>

continued...
rsrcremgr_create()

**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:

rsrcremgr_attach(), rsrcremgr_destroy()
Synopsis:

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_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 `rsrdbmgr_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 `rsrdbmgr_destroy()` function removes `count` system resources that are defined in the array `item`.

Returns:

- `EOK` Success.
- `-1` An error occurred; `errno` is set.

Errors:

- `EINVAL` Invalid argument, or the resource is in use.
- `ENOMEM` Insufficient memory to allocate internal data structures.
Examples:

See the example in `rsrcdbmgr_create()`.

Classification:

QNX Neutrino

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

See also:

`rsrcdbmgr_attach()`, `rsrcdbmgr_create()`, `rsrcdbmgr_detach()`


Synopsis:

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_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 `rsrdbmgr_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 `rsrdbmgr_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 `rsrdbmgr_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 `rsrdbmgr_attach()`.

Classification:

QNX Neutrino

| Safety                          |
|-------------------------------|----------------|
| Cancellation point            | Yes            |
| Interrupt handler             | No             |
| Signal handler                | Yes            |
| Thread                        | Yes            |

See also:

`rsrdbmgr_attach()`, `rsrdbmgr_destroy()`
Synopsis:

```c
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

dev_t rsrcdbmgr_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 `rsrcdbmgr_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>
rsrcreg_devno_attach()

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:

```c
#include <sys/rsrcreg.h>
#include <sys/rsrcregmsg.h>

char *dev_name;
int myminor_request, flags=0;
dev_t major_minor;

major_minor = rsrcreg_devno_attach
    ( dev_name, myminor_request, flags );

:

rsrcreg_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, rsrcdbmgr_attach(), rsrcdbmgr_devno_detach(), stat()
Synopsis:

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

int rsrdbmgr_devno_detach( dev_t devno,
                            int flags );
```

Arguments:

- `devno` A `dev_t` object that was returned by `rsrdbmgr_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 `rsrdbmgr_devno_detach()` detaches device number that was attached with `rsrdbmgr_devno_attach()`.

Returns:

- `EOK` Success.
- `-1` An error occurred.

Examples:

```c
#include <sys/rsrdbmgr.h>
#include <sys/rsrdbmsg.h>

dev_t dev_num;
int flags=0;
...
rsrcremgr_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:

rsrcremgr_attach(), rsrcremgr_devno_attach()
Synopsis:

```c
#include <sys/rsrcdbmgr.h>
#include <sys/rsrcdbmsg.h>

int rsrcdbmgr_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 `rsrcdbmgr_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/rsrcdbmgr.h>`):
  - RSRCDBMGR_DMA_CHANNEL — DMA channel
  - RSRCDBMGR_IOPORT — I/O port address
  - RSRCDBMGR_IRQ or RSRCMGR_IRQ — interrupt address
  - RSRCDBMGR_MEMORY — Memory address
  - RSRCDBMGR_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:

- `EINVAL` Invalid argument
- `ENOMEM` Insufficient memory to allocate internal data structures.

Examples:

List all of the memory resource blocks available in the system:

```c
rsrcref 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 2688 */
}
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()
ruserok()

Check the identity of a remote host

Synopsis:

```
#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()`, `rresvport()`

`/etc/hosts.equiv`, `rshd` in the *Utilities Reference*
C Library — S
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>ltrunc()</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>
sbrk()

Set the allocation break value

Synopsis:

```
#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 `_amblksz` (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 ) \tp\n", brk );

    brk = sbrk( 0x0200 );
    printf( "New break value after sbrk( 0x0200 ) \tp\n", brk );

    return EXIT_SUCCESS;
}
```

Classification:

Legacy Unix

Safety

Cancellation point No

continued…
sbrk()

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>

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:

`_amblksiz`, `brk()`, `_btext`, `calloc()`, `_edata`, `_end`, `errno`, `_etext`, `free()`, `malloc()`, `realloc()`


**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.
scalb(), scalbf()

Returns:

\[ x \times 2^b \]

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.

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 \times (%f \wedge %f) \n", c, a, d, b);

    return(0);
}
```

produces the output:

```
Radix of machines fp arithmetic is 2.000000
So 40.000000 = 10.000000 \times (2.000000 ^ 2.000000)
```

Classification:

\textit{scalb()} is POSIX 1003.1 XSI; \textit{scalbf()} is ANSI (draft)
### 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:

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 \times (\frac{\%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 \times (2.000000 ^ 2.000000)
```

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:

scalb()
_scalloc()
Allocate space for an array

Synopsis:

```c
#include <malloc.h>

void* scalloc( size_t size );
```

Arguments:

- size (size_t) 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
### 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(), sfree(), smalloc(), srealloc()
scandir()

Scan a directory

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:

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()`
\textbf{Synopsis:}

\begin{verbatim}
#include <stdio.h>

int scanf( const char* format, ...

\end{verbatim}

\textbf{Arguments:}

\begin{description}
\item [format] A string that controls the format of the input, as described below. The formatting string determines what additional arguments you need to provide.
\end{description}

\textbf{Library:}

\begin{verbatim}
libc
\end{verbatim}

Use the \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

\textbf{Description:}

The \texttt{scanf()} function scans input from \textit{stdin} under control of the \textit{format} argument, assigning values to the remaining arguments.

\textbf{Format control string}

The format control string consists of zero or more \textit{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:

\begin{description}
\item [multibyte characters] Any character in the \textit{format} string, other than a whitespace character or the percent character (\%), that isn’t part of a conversion specifier.
\end{description}
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 t 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, 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 by 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`. 
\textbf{scanf()}

\begin{itemize}
  \item \texttt{x, x} A hexadecimal integer, with a format as expected by \texttt{strtoul()} when \textit{base} is 16. The argument is assumed to point to an object of type \texttt{unsigned}.
  \item \texttt{[} Matches the \textit{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 by the conversion operation adds.

  When an \texttt{l} ("el") qualifier is present, a sequence of characters are converted from the initial shift state to \texttt{wchar_t} wide characters as if by a call to \texttt{mbtowc()} with \texttt{mbstate} set to 0. The argument is assumed to point to the first element of a \texttt{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 %[].)
  \item \texttt{\%} A \% character (The entire specification is \texttt{%%}).
\end{itemize}

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.

\textbf{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%hx%d", name, &hexnum, &decnum )
```

with input:

```c
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 ]%*2s%[ˆ\n]", string1, string2 );
    printf( "%s
", string1 );
    printf( "%s
", 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", weekday );
printf( "month: %s", month );
printf( "day: %d", day );
printf( "year: %d", year );

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:

fscanf(), fscanf(), sscanf(), vsscanf(), vfscanf(), vfwscanf(),
vsscanf(), vsvscanf(), vwscanf(), wscanf()
**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 `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 `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

<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, the implementation of \texttt{sched} \texttt{getparam()} isn’t 100\% POSIX 1003.1-1996. The \texttt{sched} \texttt{getparam()} function returns the scheduling parameters for thread 1 in the process \textit{pid}, or for the \textit{calling thread} if \textit{pid} is 0.

If you depend on this in new code, \textit{it will not be portable}. POSIX 1003.1 says \texttt{sched} \texttt{getparam()} should return -1 and set \texttt{errno} to \texttt{EPERM} in a multithreaded application.

See also:

\texttt{errno, getprio()}, \texttt{sched} \texttt{get priority max()}, \texttt{sched} \texttt{get priority min()}, \texttt{sched} \texttt{get scheduler()}, \texttt{sched} \texttt{param}, \texttt{sched} \texttt{setparam()}, \texttt{sched} \texttt{set scheduler()}, \texttt{sched} \texttt{yield()}, \texttt{setprio}()
Synopsis:

```
#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 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 `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() © 2006, QNX Software Systems GmbH & Co. KG.

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

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:

erro, sched_getparam(), sched_get_priority_max(), sched_get_priority_min(), sched_setparam(), sched_setscheduler(), sched_setscheduler()
sched_get_priority_max()

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

<table>
<thead>
<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_min()* , *sched_setparam()* ,
*sched_getscheduler()* , *sched_setscheduler()*


**sched_get_priority_min()**

Get the minimum priority for the scheduling policy

**Synopsis:**

```c
#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 \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.

**Library:**

```c
libc
```

Use the `-l c` 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).
**sched_get_priority_min()**

**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

**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_max()*, *sched_setparam()*, *sched_getscheduler()*, *sched_setscheduler()*
**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

<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, 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()`
Synopsis:

```c
#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;
}
```

```c
#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_req_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 *std_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,
```c
params.sched_curpriority, 
   stime, secs);
stime = etime;
itimations++;
}
prevprio = params.sched_priority;
}

return NULL;
}

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\n", 
      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 */
```
sched_param

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
**Synopsis:**

```c
#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.
sched_rr_get_interval() © 2006, QNX Software Systems GmbH & Co. KG.

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
**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.
Returns:

0   Success
-1  An error occurred (errno is set).

Errors:

EFAULT  A fault occurred trying to access the buffers provided.
EINVAL  The priority 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:

POSIX 1003.1 PS

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>

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_getscheduler(), sched_param,
sched_setscheduler(), sched_yield(), setprio()
**sched_setscheduler()**

© 2006, QNX Software Systems GmbH & Co. KG.

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.
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>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, 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_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:**

```c
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.

---

November 2, 2006  C Library — S  2743
sched_yield() © 2006, QNX Software Systems GmbH & Co. KG.

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

- Cancellation point: No
- Interrupt handler: No

continued...
**sched_yield()**

**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:**

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

Description:

The SchedCtl() and SchedCtl_r() kernel calls control the adaptive partitioning scheduler.

This scheduler is optional and is present only if you install the Adaptive Partitioning Technology Development Kit and add [module=aps] to your OS image’s buildfile. For more information, see the Adaptive Partitioning TDK 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 data argument, by calling (for example) memset(). You can also use the APS_INIT_DATA() macro:

APS_INIT_DATA( &data );

SCHED_APS_QUERY_PARMS

This command fills in a sched_aps_info structure that describes the overall parameters of the adaptive partitioning scheduler:

```c
typedef struct {
  _Uint64t cycles_per_ms;
  _Uint64t windowsize_cycles;
  _Uint64t windowsize2_cycles;
  _Uint64t windowsize3_cycles;
} sched_aps_info;
```
The members include:

- `cycles_per_ms` The number of machine cycles in a millisecond. Use this value to convert the output of the SCHED_APS_QUERY_PARTITION command to the time units of your choice.

The value of `cycles_per_ms`:

- might not equal the value of the `cycles_per_sec` member of the system page divided by 1000
- isn’t necessarily in the same units as values returned by `ClockCycles()` on all platforms

- `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.

- `windowsize2_cycles` The length, in CPU cycles, of window 2, for reporting only. Typically 10 times the window size.
windowsize3_cycles
The length, in CPU cycles, of window 3, for reporting only. Typically 100 times the window size.

scheduling_policy_flags
The set of 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_SCHEDPOL_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\_SCHEDPOL\_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 TDK 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 SIGINTR, 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.

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_policy field of the sched_apsParms structure when you call SCHED_APS_SETParms.

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_policy

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_PARTITION_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 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_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:

```c
typedef struct {
    /* input parms */
    char  *name;
    Uint16t budget_percent;
} 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 TDK User’s Guide.

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`. 

```c
(Int16t critical_budget_ms;
(Uint32t reserved1;
(Uint64t reserved2;
/* output parms */
(Int16t id;
) sched_aps_create_parms;
```
Errors:

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the
*SCHED_APS_CREATE_PARTITION* command (see the Returns section
for details):

- **EOK** Success.
- **EACCESS** `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`
  - `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:

```c
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;
    /* input parm */
    _Int16t id;
} sched_aps_partition_info;
```

The input members include:

- **id** The number of the partition you want to query.

The output members include:

- **budget_cycles** The budget, in cycles. To convert this value to something useful, convert it with the cycles_per_ms value from a SCHED_APS_QUERY_PARMS command.

- **critical_budget_cycles** The critical budget, in cycles.

- **name** The name of the partition.

- **parent_id** The number of the partition that’s the parent of the partition being queried. The System partition’s ID number is APS_SYSTEM_PARTITION_ID.
**SchedCtl(), SchedCtl_r()**

- **budget_percent**  
The partition’s budget, expressed as a percentage.

- **notify_pid, notify_tid**  
The process and thread IDs of the thread to be given overload and bankruptcy notifications, or -1 if not set.

- **pinfo_flags**  
A set of flag that give extra information about the partition:
  - SCHED_APS_PINFO_BANKRUPTCY_NOTIFY_ARMED  
    — see SCHED_APS_ATTACH_EVENTS
  - SCHED_APS_PINFO_OVERLOAD_NOTIFY_ARMED  
    — see SCHED_APS_ATTACH_EVENTS

- **pid_at_last_bankruptcy, 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:**

*_schedctl() and _schedctl_r() indicate the following errors for the 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 _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 data argument for this command must be a
sched_aps_lookup_parms structure:

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.
EINVAL  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:

```c
typedef struct {  
      _Int16t id;  
      _Int16t new_budget_percent;  
      _Int16t new_critical_budget_ms;  
      _Int16t reserved1;  
      _Int64t reserved2;  }
```

[Note: The page number mentioned in the image does not correspond to the content provided.]
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.

**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;
```
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:
SchedCtl(), SchedCtl_r()

- 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:
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:

- **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:

  \[
  \frac{100 \times idle\_cycles}{windowsize\_cycles}
  \]

- **idle_cycles_w2**: The time spent running idle during the last \(windowsize_2\_cycles\). Window 2 is typically 10 times the length of the averaging window.

- **idle_cycles_w3**: The time spent running idle during last \(windowsize_3\_cycles\). Window 3 is typically 100 times the length of the averaging window.

- **id_at_last_bankruptcy**: The ID of last bankrupt partition, or -1 if no bankruptcy has occurred.

- **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:

*SchedCtl()* and *SchedCtl_r()* indicate the following errors for the 
SCHED_APS_OVERALL_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 doesn’t match the size 
of the expected structure.
- **ENOSYS** The adaptive partitioning scheduler isn’t installed.

**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 {
    __Int32t pid;
    __Int32t tid;
    __Int32t 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.
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.
  - APS_QCRIT_RUNNING_CRITICAL — the thread is currently running as critical.
  - APS_QCRITBILL_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_QCRITBILL_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 */
    _Int16 id;
    _Int16 reserved1;
    _Int32 reserved2;
    _Int64 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`
SchedCtl(), SchedCtl_r()

- SCHED_APS_SEC_ROOT_MAKEPARTITIONS
- SCHED_APS_SEC_SYS_MAKEPARTITIONS

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.
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_securityParms 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 **SCHED\_APS\_SEC\_LOCK\_PARTITIONS** to prevent further unauthorized changes. For example:

```
sched\_aps\_security\_parms 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\_ROOT\_0\_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_SELFONLY

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_PARTITIONS_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.

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_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 `sched_aps_query_process_parms` structure:

```c
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:

- \texttt{SchedCtl()}\ EOK if the call succeeds. If an error occurs, \texttt{SchedCtl()} returns -1 and sets \texttt{errno}.

- \texttt{SchedCtl_r()}\ EOK if the call succeeds. This function \textbf{doesn’t} set \texttt{errno}. If an error occurs, \texttt{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
SchedCtl(), SchedCtl_r()

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:

SchedGet(), SchedInfo(), SchedSet(), SchedYield(), sigevent

aps in the Utilities Reference

Adaptive Partitioning TDK 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 `struct sched_param` structure where the function can store the scheduling parameters.

**Library:**

```c
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
SchedGet(), SchedGet_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>

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:
  - SCHD_FIF0 — a fixed-priority scheduler in which the highest priority, ready thread runs until it blocks or is preempted by a higher-priority thread.
  - SCHD_RR — the same as SCHD_FIF0, except threads at the same priority level timeslice (round robin) every 4 × the clock period (see ClockPeriod()).
  - SCHD_OTHER — currently the same as SCHD_RR.
  - SCHD_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()

Errors:

EINVAL    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

<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:

`sched_get_priority_max()`, `sched_get_priority_min()`, `sched_rr_get_interval()`, `SchedGet()`, `SchedSet()`
Synopsis:

```
#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 $4 \times \text{clock period}$ (see ClockPeriod()).
  - **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.
**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

<table>
<thead>
<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_setschedparam()`, `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_param`, `SchedGet()`, `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

<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:

`sched_yield()`, `SchedGet()`, `SchedSet()`
Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>

int socket( PF_INET,
            SOCK_DGRAM,
            IPPROTO_SCTP);

Or,

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_freeladdr()`, `sctp_freepaddr()`,
  `sctp_getladdr()`, `sctp_getpaddr()`, `sctp_peeloff()`, `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.
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.
**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>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:

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:**

```
#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`.
```
Classification:

SCTP

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:

SCTP, `sctp_bindx()`, `sctp_freeladdrs()`, `sctp_freepaddrs()`, `sctp_getladdrs()`, `sctp_getpaddrs()`, `sctp_peeloof()`, `sctp_recvmsg()`, `sctp_sendmsg()`
sctp_freeladdr() — Free all resources allocated by sctp_getladdr()

Synopsis:

```c
#include <netinet/sctp.h>

void sctp_freeladdr(struct sockaddr *addrs);
```

Arguments:

- `addrs` Array of local addresses returned by `sctp_getladdr()`.

Library:

- `libsctp`

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

Description:

The `sctp_freeladdr()` free all resources allocated by `sctp_getladdr()`.

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_getpaddrs()*, *sctp_peeloff()*, *sctp_recvmsg()*,
*sctp_sendmsg()*
**Synopsis:**

```c
#include <netinet/sctp.h>

void sctp_freepaddrs(struct sockaddr *addrs);
```

**Arguments:**

- `addrs`: Array of peer addresses returned by `sctp_getpaddrs()`.

**Library:**

`libsctp`

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

**Description:**

The `sctp_freepaddrs()` function frees 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>
See also:

SCTP, `sctp_bindx()`, `sctp_connectx()`, `sctp_freeladdrs()`,
`sctp_getladdrs()`, `sctp_getpaddrs()`, `sctp_peeloff()`, `sctp_recvmsg()`,
`sctp_sendmsg()`
sctp_getladdrs() © 2006, QNX Software Systems GmbH & Co. KG.

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

<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_getpaddr()`, `sctp_peeloff()`, `sctp_recvmsg()`, `sctp_sendmsg()`
Get all peer addresses in an association

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:

```
libscpt
```

Use the `-l scpt` 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:**

- **EINV AL** The passed-in address is invalid.
- **ENOMEM** Can’t allocate memory for the array of addresses.

**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_peeloff()`, `sctp_recvmsg()`, `sctp_sendmsg()`
**sctp_peeloff()**

*Branch off an association into a separate socket*

**Synopsis:**

```c
#include <netinet/sctp.h>

int sctp_peeloff(int sd,
                 sctp_assoc_t assoc_id);
```

**Library:**

`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()**

Receive a message, using advanced SCTP features

**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:**

`libsctp`

Use the `-l sctp` option to `qcc` to link against this library.
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_freeladdrx(),
    sctp_freeaddrs(), sctp_getladdrx(), sctp_getaddrs(), sctp_peeloff(),
    sctp_sendmsg()
Synopsis:

```c
#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 \texttt{-l sctp} option to \texttt{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.
- **ENOBIFS** 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()**

<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_freeladdrs()`, `sctp_freepaddrs()`, `sctp_getladdr()`, `sctp_getpaddrs()`, `sctp_peeloff()`, `sctp_recvmsg()`,
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:

- **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()  © 2006, QNX Software Systems GmbH & Co. KG.

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

Cancellation point    No
Interrupt handler     No
Signal handler        No
Thread                Yes
Caveats:

The `searchenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

`getenv()`, `pathfind()`, `pathfind_r()`, `setenv()`
seed48()  Initialize the seed for a sequence of pseudo-random numbers

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 \( a = 0x\text{FDEECE66D} = 25214903917 \) 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
seed48()

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:

`drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `mrand48()`, `nrand48()`, `srand48()`
**seekdir()**

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()`.
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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</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.

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: \( t_{sec} \) must be between 0 and 1\(^{0}8\), inclusive; \( t_{usec} \) 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;

    return select( 2, &rfd, 0, 0, &tv );
}
```

November 2, 2006  C Library — S  2827
```
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**

<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>Read the Caveats</td>
</tr>
</tbody>
</table>

**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.
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()`
select_attach()

Attach a file descriptor to a dispatch handle

Synopsis:

```c
#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:

  ```c
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_attach()**

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.
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>
<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:

`select_detach()`, `select_query()`
select_detach()

Detach a file descriptor from a dispatch handle

Synopsis:

```c
#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:

```c
#include <sys/dispatch.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int my_func( ... ) {
    ...
```
```c
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 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 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**

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

*continued...*
select_detach()

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

See also:

select_attach(), select_query()
**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()

© 2006, QNX Software Systems GmbH & Co. KG.

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_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>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>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:

```c
#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`.

**Note**

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`. 
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_init()`, `sem_open()`, `sem_unlink()`

`procnto*` in the *Utilities Reference*
sem_destroy()  
Destroy a semaphore

Synopsis:

```
#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 (EINVVAL).

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:
POSIX 1003.1 SEM

Safety
Cancellation point  No
Interrupt handler  No
Signal handler  Yes
Thread  Yes

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:

`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).
Errors:

EINV AL 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:

```c
#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.
- `ENOSPC` A resource required to initialize the semaphore has been exhausted.
- `ENOSYS` The `sem_init()` function isn’t supported.

**Classification:**

POSIX 1003.1 SEM

**Safety**

- Cancellation point: No
- Interrupt handler: No

continued...
**sem_init()**

---

**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>

**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()

Create or access a named semaphore

Synopsis:

```c
#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.
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.

EINTR The call was interrupted by a signal.

ELOOP Too many levels of symbolic links or prefixes.

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 procnot is 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.
sem_open()

Classification:

POSIX 1003.1 SEM

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:

sem_close(), sem_destroy(), sem_init(), sem_post(), sem_trywait(),
sem_unlink(), sem_wait()

mqueue, procnto* in the Utilities Reference
Synopsis:

```c
#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()**

**Errors:**

EINVAL     Invalid semaphore descriptor `sem`.

ENOSYS     The `sem_post()` function isn’t supported.

**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:**

`errno`, `sem_destroy()`, `sem_init()`, `sem_trywait()`, `sem_wait()`
sem_timedwait()  
Wait on a semaphore, with a timeout

Synopsis:
```c
#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:
```c
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.
**sem_timedwait()**

**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 );
```

November 2, 2006

© 2006, QNX Software Systems GmbH & Co. KG.
Semaphore acquired after %d timeouts
return;
}

Classification:

POSIX 1003.1 SEM TMO

<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_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:

```c
#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 *sem_name* doesn’t exist.

ENAMETOOLONG The *sem_name* argument is longer than (NAME_MAX - 8).

ENOSYS The *sem_unlink()* function isn’t implemented for the filesystem specified in *path*.

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:

*sem_open(), sem_close(), sem_wait(), sem_trywait(), sem_post()*

*procnto* in the *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`.
- `EINVAL` 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()`
Synopsis:

```c
#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.
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.
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.

EWOULDBLOCK

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()
Send a message and its header to a socket

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.
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:

getsockopt(), ioctl(), recv(), select(), send(), sendto(), socket(), write()
sendto()

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.
- **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.

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 gcc 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>
</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(), 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
**setbuf()**

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 BUFSIZE 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( BUFSIZE );
    if( buffer == NULL ) {
        return EXIT_FAILURE;
    }
    return 0;
}
```

November 2, 2006
```c
fp = fopen( "some_file", "r" );
setbuf( fp, buffer );
/*.*/*
/*.*/*
/*.*/*
fclose( fp );
free( buffer );
return EXIT_SUCCESS;
}
```

**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:**

`fopen()`, `setvbuf()`
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:

```
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>
</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:**

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()`
setegid()

Set the effective group ID for a process

Synopsis:

```c
#include <unistd.h>

int setegid( gid_t gid );
```

Arguments:

- `gid_t gid` The effective group ID that you want to use for the process.

Library:

```c
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.

Advertisement

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()

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:

/*
 * 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

Cancellation point   No

continued…
setegid()

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:

`errno`, `getegid()`, `seteuid()`, `setgid()`, `setuid()`
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;
}
```
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`, `exec()`, `execl()`, `execlp()`, `execlpe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `searchenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnvp()`, `spawnve()`, `spawnvpe()`, `system()`, `unsetenv()`
seteuid()  © 2006, QNX Software Systems GmbH & Co. KG.
Set the effective user ID

Synopsis:

#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.
seteuid()

Returns:

0  Success.

-1  An error occurred (errno is set).

Errors:

EINV AL  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 the saved set-user ID.

Examples:

/*
 * 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;
}
seteuid()

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:

erno, geteuid(), setegid(), setuid(), setgid()
setgid()
Set the real, effective and saved group IDs

Synopsis:
#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 AL 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

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:

errno, setegid(), seteuid(), setuid()
**setgrent()**  
**Rewind to the start of the group database file**

**Synopsis:**

```c
#include <grp.h>

int setgrent( void );
```

**Library:**

```c
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

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
</table>

*continued...*
setgrent()

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:

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`.
setgroups()

Classification:

Unix

Safety

<table>
<thead>
<tr>
<th>Category</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:

groups(), initgroups()
sethostent()
Open the host database file

Synopsis:

```c
#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

<table>
<thead>
<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:

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() © 2006, QNX Software Systems GmbH & Co. KG.
Set the name of the current host

Synopsis:

```c
#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).
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

<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:
This function is restricted to the superuser, and is normally used only at boot time.

See also:

`gethostname()`
**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
SETIOV()

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>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(), GETIOVLEN(), MsgKeyData(), MsgReadv(),
MsgReceivev(), MsgReplyv(), MsgSendv(), MsgWritev()
setitimer()  © 2006, QNX Software Systems GmbH & Co. KG.
Set the value of an interval timer

Synopsis:

#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:

struct timeval it_interval; /* timer interval */
struct timeval it_value; /* current value */
The \textit{it\_value} member indicates the time to the next timer expiration. The \textit{it\_interval} member specifies a value to be used in reloading \textit{it\_value} when the timer expires. Setting \textit{it\_value} to 0 disables a timer, regardless of the value of \textit{it\_interval}. Setting \textit{it\_interval} to 0 disables a timer after its next expiration (assuming \textit{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 \texttt{ITIMER\_REAL}, which decrements in real time. A \texttt{SIGALRM} signal is delivered when this timer expires.

The \texttt{SIGALRM} so generated isn’t maskable on this bound thread by any signal-masking function, \texttt{pthread\_sigmask()}, or \texttt{sigprocmask()}.

**Returns:**

- \texttt{0} Success.
- \texttt{-1} An error occurred; \texttt{errno} is set.

**Errors:**

- \texttt{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 \textit{which} argument is unrecognized.

**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>
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:

```
#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.
Returns:

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 = set jmp( env );
    if( ret_val == 0 )
    {
        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 output:

```
after set jmp(): 0
about to long jmp()
back from long jmp(): 14
```
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:

`longjmp()`
setkey()  
Set the key used in encryption

Synopsis:

```c
#include <stdlib.h>

void setkey( const char *key );
```

Arguments:

`key`  
A 64-character array of binary values (numeric 0 or 1).

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 `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

<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>No</td>
</tr>
</tbody>
</table>
See also:

`crypt()`, `encrypt()`
**setlinebuf()**

Assign line buffering to a stream

**Synopsis:**

```c
#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:

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:

A pointer to a system-generated string indicating the previous locale, or NULL if an error occurs.

Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
</table>

continued...
setlocale()

© 2006, 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>Yes</td>
</tr>
</tbody>
</table>

See also:

isalpha(), isascii(), localeconv(), printf(), scanf(), strcat(), strchr(), strcmp(), strcoll(), strcpy(), strftime(), strlen(), strpbrk(), strspn(), strtol(), strtok(), strxfrm(), tm
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

<table>
<thead>
<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:
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
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:

`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.
setpgid()

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:

```c
/*
 * 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

<table>
<thead>
<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, setpgid(), getsid(), setsid()`
Synopsis:

```c
#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>
See also:

exec(), execve(), execl(), execlp(), execv(), execvp(), execvpe(), fork(), getpid(), getpgrp(), getsid(), kill(), signal()
setprio()  Set the priority of a process

Synopsis:

#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.
setprio()

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:

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 `getprotobynumber()` 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

Safety

- Cancellation point: Yes
- Interrupt handler: No

continued...
setprotoent()

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>

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(), getprotobyname(), getprotobynumber(), getprotoent(), protoent

/etc/protocols in the Utilities Reference
**setpwent()**

Rewind the password database file

**Synopsis:**
```
#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()
Set real and effective group IDs

Synopsis:

#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. rgid 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 rgid 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

<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:

execve(), getgid(), setreuid(), setuid()
setreuid() Set real and effective user IDs

Synopsis:

```c
#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:

EINVAL    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>
</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:

`execve()`, `getuid()`, `setregid()`, `setuid()`
setrlimit(), setrlimit64()
Set the limit on a system resource

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 `gcc` 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,</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>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<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>
<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_NOFILE</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>
</tbody>
</table>

*continued...*
### Resource Description Action

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_NPROC</td>
<td>The maximum number of processes.</td>
<td>Attempts to create new processes fail.</td>
</tr>
<tr>
<td>RLIMIT_NTHR</td>
<td>The maximum number of threads.</td>
<td>Attempts to create threads (e.g. by calling <code>pthread_create()</code>) 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.</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></td>
<td>Within a process, setrlimit() 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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Within a multithreaded process, setrlimit() has no impact on the stack size limit for the calling thread if the calling thread isn’t the main thread. A call to setrlimit() for RLIMIT_STACK impacts only the main thread’s stack, and should be made only from the main thread, if at all.</td>
<td></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 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 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

<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(), execlpe(), execv(), execve(), execvp(), execvepe(), fork(), getdtablesize(), getrlimit(), getrlimit64(), malloc(), open(), signal(), sysconf()

ulimit builtin command (see the entry for ksh in the Utilities Reference)
setservent()  © 2006, QNX Software Systems GmbH & Co. KG.

Open the network services database file

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:

```c
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

<table>
<thead>
<tr>
<th>Safety</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>
</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>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:

endservent(), getservbyname(), getservbyport(), getservent(),

/etc/services in the Utilities Reference.
**setsid()**

Create a new session

**Synopsis:**

```c
#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:**

```c
/*
 * 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.
 */
```
```c
#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**

<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, setsid(), setpgid()`
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_SNDBUF, 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>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:

- **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.
- **EINVAL**: No `optval` value was specified.
- **ENOPROTOOPT**: The option is unknown at the level indicated.

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></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>continued...</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
setsockopt()

**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:**

ICMP, IP, TCP, and UDP protocols

getsockopt(), socket()
setspent() 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
```

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:

```
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>
<th></th>
</tr>
</thead>
<tbody>
<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()`, `initstate()`, `rand()`, `random()`, `srand()`, `srandom()`
settimeofday()
Set the time and date

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:

`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).
settimeofday()

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:
#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.

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.

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()

© 2006, QNX Software Systems GmbH & Co. KG.

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

Cancellation point No

continued…
See also:

errno, getuid(), setgid(), seteuid(), setgid()
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.

Note: 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
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.
Returns:

0              Success.
EINV AL        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
setvbuf()

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:

fopen(), setbuf()
Synopsis:
#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
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>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()`, `smalloc()`, `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.
shm_ctl()

Library:

`libc`

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

Description:

The `shm_ctl()` function modifies the attributes of the shared memory object identified by the handle, `fd`. This handle is the value returned by `shm_open()`.

The combination `SHMCTL_ANON | SHMCTL_PHYS` has the same behavior as for `mmap()`: it indicates that you want physically contiguous RAM to be allocated for the object.

The `shm_ctl_special()` function is similar to `shm_ctl()`, but has an additional, processor-specific argument.

Flags for ARM processors

For an ARM processor, the behavior for different combinations of `flags` arguments is as follows:

- `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 `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.

**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
#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 \"
```
The following is meant to be run with `sharephyscreator`.

```
/*
 * sharephysuser.c
 *
November 2, 2006 C Library — S 2973
```
This one is meant to be run in tandem with sharephyscreator.

* Run it as: sharephysuser shared_memory_object_name length
* Example: sharephysuser wally 49
* */

```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/stat.h>

char *progname = "sharephysuser";

int main( int argc, char *argv[] )
{
    int fd, len, i;
    char *ptr, *name;
    if ( argc != 3 ) {
        fprintf( stderr, "Use: sharephysuser shared_memory_object_name \n length\n" );
        fprintf( stderr, "Example: sharephysuser wally 49\n" );
        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\n", 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\n", progname, strerror(errno) );
        exit( EXIT_FAILURE );
    }
    printf( "%s: reading the text: ", progname );
    for ( i = 0; i < len; i++ )
        printf("%c", ptr[i] );
    printf( "\n" );
    close( fd );
    munmap( ptr, len );
}
```

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>

char *progname = "sharephysuser";

int main( int argc, char *argv[] )
{
    int fd, len, i;
    char *ptr, *name;
    if ( argc != 3 ) {
        fprintf( stderr, "Use: sharephysuser shared_memory_object_name \n length\n" );
        fprintf( stderr, "Example: sharephysuser wally 49\n" );
        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\n", 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\n", progname, strerror(errno) );
        exit( EXIT_FAILURE );
    }
    printf( "%s: reading the text: ", progname );
    for ( i = 0; i < len; i++ )
        printf("%c", ptr[i] );
    printf( "\n" );
    close( fd );
    munmap( ptr, len );
}
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:

`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.
  - `SHMCTLLOWERPROT` — 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`. 
**shm_ctl_special()**

`size` The new size of the object, in bytes, regardless of ANON/PHYS flag.

`special` 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:**

EINVVAL An invalid combination of flags was specified, or the shared memory object is already “special.”
shm_ctl_special()

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:

`mmap()`, `munmap()`, `mprotect()`, `shm_ctl()`, `shm_open()`, `shm_unlink()`, `ThreadCtl()`
Synopsis:

```c
#include <fcntl.h>
#include <sys/mman.h>

int shm_open( const char * name,
              int oflag,
              mode_t mode );
```

Arguments:

- **name**: The name of the shared memory object that you want to open; see below.
- **oflag**: 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 `shm_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 `shm_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()`.

Open a shared memory object
Library:

**libc**

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 file descriptor 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><code>CWD/entry</code></td>
</tr>
<tr>
<td><code>/entry</code></td>
<td><code>/dev/shmem/entry</code></td>
</tr>
<tr>
<td>entry/newentry</td>
<td><code>CWD/entry/newentry</code></td>
</tr>
<tr>
<td><code>/entry/newentry</code></td>
<td><code>/entry/newentry</code></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.
shm_open()

**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 oflag are denied, or O_TRUNC is specified and write permission is denied.

- **EEXIST**  
  O_CREAT and O_EXCL are set, and the named shared memory object already exists.

- **EINVAL**  
  The 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 name argument exceeds NAME_MAX.

- **ENFILE**  
  Too many shared memory objects are currently open in the system.

- **ENOENT**  
  O_CREAT isn’t set, and the named shared memory object doesn’t exist, or O_CREAT is set and either the name prefix doesn’t exist or the name argument points to an empty string.

- **ENOSPC**  
  There isn’t enough space to create the new shared memory object.

- **ENOSYS**  
  The 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:

```c
#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;
    }

    /* In case the unlink code isn’t executed at the end */
    if( argc != 1 ) {
        shm.unlink( "/bolts" );
        return EXIT_SUCCESS;
    }
}
```


This example uses a shared memory object to share data with a forked process:

```c
#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;
}

/* 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;
Classification:

POSIX 1003.1 SHM

<table>
<thead>
<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:

`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).
shm_unlink()

Errors:

- **EACCES**  Permission to unlink the shared memory object is denied.
- **ELOOP**  Too many levels of symbolic links or prefixes.
- **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

<table>
<thead>
<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(), mprotect(), shmctl(), shmctl_special(), shm_open()
**shutdown()**

Shut down part of a full-duplex connection

**Synopsis:**
```
#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:
    - SHUT_RD Further receives
    - SHUT_WR Further sends
    - SHUT_RDWR Further sends and receives

**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).
shutdown()

Errors:

EBADF Invalid descriptor s.

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:

close(), connect(), socket()
**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:
**sigaction()**

*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-catch 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).
**sigaction()**

- 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:

```c
void handler(int signo, siginfo_t *info, void *other)
```

If you have an old-style signal handler of the form:

```c
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:

```c
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 \texttt{kill()} function generated the signal.
- SI_QUEUE — the \texttt{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 \texttt{SignalKill()}). When targeted at a process, at most one thread receives the signal. This thread must have the signal unblocked (see \texttt{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 \texttt{longjmp()} to return from a signal handler, the signal remains masked. You can use \texttt{siglongjmp()} to restore the mask to the state saved by a previous call to \texttt{sigsetjmp()}. 
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 )
```


{ 
    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() unmask 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

<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:

erro, kill(), raise(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(),
sigismember(), signal(), SignalAction(), SignalKill(), sigpending(),
sigprocmask()
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()`.

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()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal()`, `sigpending()`, `sigprocmask()`
**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:

```
#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()`
sigemptyset() © 2006, QNX Software Systems GmbH & Co. KG.

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 )
```

3002 C Library — S November 2, 2006


```c
{
    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;
}
```

**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>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`kill()`, `raise()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigfillset()`, `sigismember()`, `signal()`, `sigpending()`, `sigprocmask()`
Structure that describes an event

Synopsis:

```c
#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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(special — see below)</td>
<td></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 `sigevent` structure. Don’t compare this member 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.
**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_MINA VAIL` to `_PULSE_CODE_MAXA VAIL`.

**void *sigev_value.sival_int**

A 32-bit value to be interpreted by the pulse handler.

The initialization macro is:

`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:

`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:

`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_MINAVAL through SI_MAXAVAL.

- **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_MINAVAL through SI_MAXAVAL.

- **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 )
```
SIGEV_THREAD

Create a new thread.

We don’t recommend using this type of event. Pulses are more efficient.

The following fields are used:

\[
\text{void (*sigev.notify\_function)} (\text{union sigval})
\]

A pointer to the function to be notified.

\[
\text{pthread\_attr \*sigev.notify\_attributes}
\]

A pointer to thread attributes. This must be NULL, or point to a structure initialized by \text{pthread\_attr\_init()} at the time of delivery.

\[
\text{void \*sigev.value.sival\_ptr}
\]

A value that’s to be passed to the notification function.

The initialization macro is:

\[
\text{SIGEV\_THREAD\_INIT( } \text{&event, fn, value, attr })
\]

SIGEV_UNBLOCK

Force a thread to become unblocked. No other fields in the structure are used.

The initialization macro is:

\[
\text{SIGEV\_UNBLOCK\_INIT( } \text{&event })
\]

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. In order to use adaptive partitioning, you need to install the Adaptive Partitioning Technology Development Kit. 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:

```c
SIGEV_MAKE_CRITICAL(&event)
```
Make the targeted thread run as critical.

```c
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:**

- `ds_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
Adaptive Partitioning *User’s 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

Safety

- Cancellation point: No
- Interrupt handler: Yes

continued...
### sigfillset()

© 2006, 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()`
sigismember()  
See if a given signal is in a given set

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:

```c
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:

```c
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:

```
#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>
</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:

`longjmp()`, `sigaction()`, `sigprocmask()`, `sigsuspend()`
**sigmask()**

**Construct a mask for a signal number**

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

#define sigmask(s) (1L<<(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

<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...*
sigmask()

Safety

| Thread | Yes |

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:

```c
#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:
/* "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 )
```

November 2, 2006
signal()

{  
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

<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_sub_value(),  
atomic_toggle(), atomic_toggle_value(), InterruptLock(),  
InterruptUnlock(), kill(), longjmp(), raise(), siglongjmp(),  
sigprocmask()
**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.
**SignalAction(), SignalAction_r()**

© 2006, QNX Software Systems GmbH & Co. KG.

### 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.

☞ 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.

```c
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:

```c
void (*sa_handler)();
```

The address of a signal handler or action for nonqueued signals.

```c
void (*sa_sigaction)(int signo, siginfo_t *info, void *other);
```

The address of a signal handler or action for queued signals.

```c
sigset_t sa_mask
```

An additional set of signals to be masked (blocked) during execution of the signal-catching function.

```c
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.
cdot 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:

```c
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.
SignalAction(), SignalAction_r() © 2006, QNX Software Systems GmbH & Co. KG.

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

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:

abort(), ChannelCreate(), kill(), longjmp(), siglongjmp(), signal(), sigaction(), SignalKill(), SignalProcmask(), sigqueue(), sigsetjmp(), SyncMutexLock(), ThreadCreate(), wait(), waitpid()
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:

libc

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

Description:

The SignalKill() and SignalKill_r() kernel calls send the signal signo with a code specified by code and a value specified by 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 signo is zero, no signal is sent, but the validity of pid and tid are checked. You can use this as a test for existence.

SignalKill() implements the capabilities of the POSIX functions kill(), sigqueue(), and pthread_kill() in one call. Consider using these functions instead of invoking these kernel calls directly.

The pid and tid determine the target of the signal, as follows:

<table>
<thead>
<tr>
<th>pid</th>
<th>tid</th>
<th>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 -pid</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>= 0</td>
<td>Hit a single process identified by pid</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>&gt; 0</td>
<td>Hit a single thread in process pid identified by 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 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 SignalProcmask(), SignalSuspend(), and 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 signo       The signal number.
int code       The signal code.
union sigval val
               The signal value.
```

The value of `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;= 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 \textit{signo}, \textit{code}, and \textit{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:

\begin{align*}
\text{\textit{SignalKill}()} & \quad \text{If an error occurs, -1 is returned and sets } \textit{errno}. \\
& \quad \text{Any other value returned indicates success.} \\
\text{\textit{SignalKill\_r}()} & \quad \text{EOK is returned on success. This function does NOT set } \textit{errno}. \text{ If an error occurs, any value in the Errors section may be returned.}
\end{align*}

**Errors:**

\begin{itemize}
\item **EINVAL** \quad The value of \textit{signo} is less than 0 or greater than \textit{(_NSIG -1)}.
\item **ESRCH** \quad The process or process group indicated by \textit{pid} or thread indicated by \textit{tid} doesn’t exist.
\item **EPERM** \quad The process doesn’t have permission to send the signal to any receiving process.
\item **EAGAIN** \quad The kernel had insufficient resources to enqueue the signal.
\end{itemize}
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_kill(), kill(), SignalAction(), SignalProcmask(), SignalSuspend(), SignalWaitinfo(), sigqueue()
SignalProcmask(),
SignalProcmask_r()

Modify or examine the signal-blocked mask of a thread

Synopsis:

```
#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.
SignalProcmask(), SignalProcmask_r()

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_sigmask(), SignalAction(), SignalKill(), sigprocmask()
**SignalSuspend(), SignalSuspend_r()** © 2006, QNX Software Systems GmbH & Co. KG.

*Suspend a thread until a signal is received*

**Synopsis:**

```c
#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.

**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

<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()`
SignalWaitinfo(), SignalWaitinfo_r()
Select a pending signal

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:

<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.
SignalWaitinfo() , SignalWaitinfo_r()

Errors:

EINTR  The call was interrupted by a signal.
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

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:

SignalKill() , SignalKill_r() , sigwaitinfo()
**significand(), significandf()**  © 2006, QNX Software Systems GmbH & Co. KG.

Compute the “significant bits” of a floating-point number

**Synopsis:**

```c
#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 \(\text{sign} \times 2^n\) with \(1 < \text{sign} < 2\), then `significand(x)` returns `sign` 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`.
significand(), significandf()

Returns:

\[
scalb \left( x, (\text{double})^{-\text{ilogb}}(x) \right)
\]

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 \( x = a \times 2^n \), then:

When \( x \) is: \( \text{scalbn}(x, n) \) returns:

<table>
<thead>
<tr>
<th>( \pm \infty )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{NAN} )</td>
<td>( \text{NAN} )</td>
</tr>
</tbody>
</table>

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 * (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)
significand(), significandf()

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:

ilogb(), scalb(), scalbn()
sigpause()  
Wait for a signal

Synopsis:

```
#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.
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>

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()`
**Synopsis:**

```
#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()
sigprocmask()

Examine or change the signal mask for a thread

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.

November 2, 2006 C Library — S 3055
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 %8ld.\n", oset );

sigpending( &pset );
printf( "Pending set is %8ld.\n", pset );
kill( getpid(), SIGINT );

sigpending( &pset );
printf( "Pending set is %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()
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:

```
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:

  
  ```
  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()**

*Saves 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()`.

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()`
**Synopsis:**

```c
#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.
sigsetmask()

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(), 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-catch 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.
sigsuspend()

Examples:

/*
 * This program pauses until a signal other than
 * a SIGINT occurs. In this case a SIGALRM.
 */
#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"
    " 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

<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>

November 2, 2006
See also:

pause(), pthread_sigmask(), sigaction(), SignalWaitinfo(),
sigpending(), sigsuspend(), sigwaitinfo(), timespec
**sigunblock()**

Unblock signals

**Synopsis:**
```
#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()
Synopsis:

```c
#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:

0  Success (that is, one of the signals specified by set is pending or has been generated).
EINTR  The sigwait() function was interrupted by a signal.
EINVAL  The set argument contains an invalid or unsupported signal number.
EFAULT  A fault occurred while accessing the provided buffers.

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:

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.
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

<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:

pause(), pthread_setmask(), sigaction(), SignalWaitinfo(), sigpending(), sigsuspend(), sigtimedwait(), sigwait()
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 )
{
```
```c
    printf( "%f\n", sin(.5) );
    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()`
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.

 brewers  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 )
```
\texttt{sinh()}, \texttt{sinhf()}

{ 
  \texttt{printf( \"%f\n\", sinh(.5) );}

  \texttt{return( EXIT_SUCCESS );}
}

produces the output:

0.521095

\textbf{Classification:}

ANSI, POSIX 1003.1

\begin{tabular}{|l|l|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\hline
\end{tabular}

\textbf{See also:}

\texttt{cosh()}, \texttt{errno}, \texttt{tanh()}

3080 C Library — S November 2, 2006
sleep()  
Suspend a thread for a given length of time

Synopsis:

```c
#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:

```c
#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()`


## Synopsis:

```c
#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>
<th></th>
</tr>
</thead>
<tbody>
<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_destroy(), _sleepon_init(), _sleepon_lock(), _sleepon_signal(),
_sleepon_unlock()
Synopsis:

```
#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
_slepon_destroy()

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:

_slepon_broadcast(), _slepon_init(), _slepon_lock(),
_slepon_signal(), _slepon_unlock()
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
_slepon_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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

_slepon_broadcast(), _slepon_destroy(), _slepon_lock(),
_slepon_signal(), _slepon_unlock()


**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

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:

pthread_mutex_lock(), sleepon_broadcast(), sleepon_destroy(),
sleepon_init(), sleepon_signal(), sleepon_unlock(), sleepon_wait()
Synopsis:

```c
#include <pthread.h>

int _sleepon_signal( 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 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()`.
_sleep_on_signal()"

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_sleep_on_broadcast(), pthread_sleep_on_signal(),
sleep_on_broadcast(), sleep_on_destroy(), sleep_on_init(),
sleep_on_lock(), sleep_on_unlock()
_sleepon_unlock()
Unlock a sleepon

Synopsis:

```c
#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
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>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()
Synopsis:

```c
#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()`.
_sleepon_wait()

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()
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.

- `Note` 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 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>
<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:

`slogf(), slogf(), vslogf()`

slogger, sloginfo in the Utilities Reference
**Synopsis:**

```c
#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).
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>
<tr>
<td>_SLOG_DEBUG1</td>
<td>6</td>
<td>Debug messages (normal detail)</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_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(_SLOG_TEST, 0),
                      _SLOG_DEBUG1, &i, sizeof(i));
                break;
            case 1:
                slogi(_SLOG_SETCODE(_SLOG_TEST, 1),
                      _SLOG_CRITICAL, 1, i);
                break;
            case 2:
                slogf(_SLOG_SETCODE(_SLOG_TEST, 2),
                      _SLOG_ERROR,
                      "This is number %d", i);
        }
    }
}
```

November 2, 2006
slogf()

break;
}

sleep(1);
}

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:
slogb(), slogi(), vslogf()
slogger, sloginfo in the Utilities Reference
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 `-lc` 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 \textit{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 \textit{slogger} isn’t running.

Examples:

See \textit{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:

\textit{slogb(), slogf(), slogi(), vslogf()}

\textit{slogger, sloginfo} in the \textit{Utilities Reference}
**_smalloc()**

**Allocate memory in blocks**

**Synopsis:**

```c
#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.
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:

`libsnmp`

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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**Classification:**

SNMP

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Yes</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>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`

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

<table>
<thead>
<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

Marshall T. Rose, The Simple Book: An Introduction to Internet
0-13-451659-1)
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

**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:**

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:

```c
#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.
srcParty  The source party being used.
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.
dstPartyLen  The number of OID elements in dstParty.
context  The context being used.
contextLen  The number of OID elements in context.
community  The community for outgoing requests.
communityLen  The length of the community name.
command  The type of this PDU.
reqid  The request ID. The default is SNMP_DEFAULT_REQID (0).
errstat  The error status (non_repeaters in GetBulk). The default is SNMP_DEFAULT_ERRSTAT (-1).
errindex  The error index (max_repetitions in GetBulk). The default is SNMP_DEFAULT_ERRINDEX (-1).
enterprise  The system OID.
enterpriseLength  The number of OID elements in enterprise. The default is SNMP_DEFAULT_ENTERPRISE_LENGTH (0).
agent_addr  The address of the object generating the trap.
trap_type  The trap type.
specific_type  The specific type.
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:

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`

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:

- `libsnp`  
  
  Use the `-l snmp` option to `qcc` to link against this library.
snmp_pdu_create()

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 (`snmp_errno` is set).

Errors:
If an error occurs, this function sets `snmp_errno` to:

SNMPERR_GENERR
Not enough memory to create the `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:

`snmp_free_pdu()`, `snmp_pdu`, `snmp_read()`, `snmp_send()`

RFC 1157, FAQ in Internet newsgroup `comp.protocols.snmp`

November 2, 2006
**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

<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...*
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)
Get information that select() needs for SNMP

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:**

```bash
libsnp
```

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

**Description:**

The `snmp_select_info()` function is used to return information about what SNMP requires from a `select()` call.

**Asynchronous SNMP transactions:**

To have SNMP transactions occur asynchronously, you can invoke the functions `snmp_select_info()`, `snmp_timeout()`, and `snmp_read()` in conjunction with the system call `select()`. For more information, see `select()`.

For asynchronous transactions, invoke `snmp_select_info()` with the information you would have passed to `select()` in the absence of SNMP. The `snmp_select_info()` function modifies the information, which is subsequently passed to `select()`.

<table>
<thead>
<tr>
<th>Parameters to <code>select()</code></th>
<th>Corresponding parameters to <code>snmp_select_info()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nfds</code></td>
<td><code>numfds</code></td>
</tr>
<tr>
<td><code>readfds</code></td>
<td><code>fdset</code></td>
</tr>
<tr>
<td><code>timeout</code></td>
<td><code>timeout</code>——must point to an allocated (but not necessarily initialized) <code>timeval</code> structure.</td>
</tr>
</tbody>
</table>

The following code segment shows how to use these SNMP functions in conjunction with `select()`:

```c
FD_ZERO(&fdset);
numfds=sd+1;
FD_SET(sd, &fdset);
block=0;
tvp=&timeout;
```
snmp_select_info()

```
timerclear(tvp);
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>
</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()`, `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_send()**  
Send SNMP messages

**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:**

`libsnpmp`

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

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).
### snmp_session

**peernam**

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:

```c
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.
snmp_session

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstPartyLen</td>
<td>The number of OID elements in dstParty.</td>
</tr>
<tr>
<td>context</td>
<td>The context being used for this session.</td>
</tr>
<tr>
<td>contextLen</td>
<td>The number of OID elements in context.</td>
</tr>
</tbody>
</table>

Classification:

SNMP

See also:

snmp_close(), snmp_free_pdu(), snmp_open(), snmp_pdu, snmp_send()

RFC 1157, FAQ in Internet newsgroup comp.protocols.snmp

**Synopsis:**

```c
#include <snmp/snmp_api.h>

void snmp_timeout( void );
```

**Library:**

```
libsnp
```

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>
<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>

November 2, 2006
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

`snprintf()`

Write formatted output to a character array, up to a given maximum number of characters

**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 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()`.

**Library:**

```c
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
snprintf() 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>
</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>

**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()`, `vfprintf()`, `vfwprintf()`, `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).
sockatmark()

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
socket()

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:

By default, socket() communicates with the TCP/IP stack managing the /dev/socket portion of the namespace. This behavior can be controlled via the SOCK environmental variable. See the examples in the npm-tcpipe.so utility.

See also:

ICMP6, ICMP, INET6, IPv6, IP, IPsec, ROUTE, TCP, UDP, UNIX protocols

accept(), bind(), close(), connect(), getprotobyname(), getsockname(),
getsockopt(), ioctl(), listen(), read(), recv(), select(), send(),
shutdown(), socketpair(), write()
socketpair() © 2006, QNX Software Systems GmbH & Co. KG.
Create a pair of connected sockets

Synopsis:

```c
#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.
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");
```
socketpair()

```c
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

<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()`
**SOCKSinit()**

Initialize a connection with a SOCKS server

**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
SOCKSinit()© 2006, 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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

**See also:**

`Raccept()`, `Rbind()`, `Rconnect()`, `Rgetsockname()`, `Rlisten()`, `Rcmd()`, `Rselect()`

SOCKS — A Basic Firewall
Synopsis:

```c
#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).
The shared access for the file. This is a combination of the following bits (defined in `<share.h>`):

- 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

```c
open(path, oflag, ...);
```

is the same as:

```c
sopen(path, oflag, SH_DENYNO, ...);
```

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

<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...
sopen()

Safety
Thread       Yes

See also:
chsize(), close(), creat(), dup(), dup2(), eof(), execl(), execlp(),
execvp(), execvp(), fcntl(), 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_DENYNO` — 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()

<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:

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>Description</th>
</tr>
</thead>
<tbody>
<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, 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)
- 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.
- 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()

- SPAWN_CHECK_SCRIPT — if path starts with 
  #!, 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**  
The child process’s processor affinity - the ability to associate the process with a particular processor.

This member was added in the QNX Neutrino Core OS 6.3.2.
**spawn()**

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.

uint32_t nd

The node descriptor of the remote node on which to spawn the child process. This member is used.
spawn()

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()**

<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:**

execl(), execlp(), execvpe(), execvp(), execlp(), execvp(), execvpe(), getenv(), netmgr_strtond(), putenv(), sched_param, setenv(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), spawnl(), spawnle(), spawnlpe(), spawnlp(), spawnvp(), spawnvpe(), spawnvp(), spawnvpe(), wait(), waitid()
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>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 <em>waitpid()</em> 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.
- **EFAULT** One of the buffers specified in the function call is invalid.
spawnl()

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 <stdio.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

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Read the 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:**

If `mode` is `P_WAIT`, this function is a cancellation point.

**See also:**

- `exec()`,
- `execl()`,
- `execlp()`,
- `execle()`,
- `execvp()`,
- `execvpe()`,
- `getenv()`,
- `putenv()`,
- `setenv()`,
- `spawn()`,
- `spawnle()`,
- `spawnlp()`,
- `spawnlpe()`,
- `spawnp()`,
- `spawnv()`,
- `spawnve()`,
- `spawnvp()`,
- `spawnvpe()`,
- `wait()`,
- `waitpid()`
spawnle()

Spawn a child process, given a list of arguments and an environment

**Synopsis:**

```
#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:**

```
libc
```

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.
spawnle()

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 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:
spawnle()

```c
#include <stddef.h>
#include <process.h>

char *env_list[] = { "SOURCE=MYDATA",
                     "TARGET=OUTPUT",
                     "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**

<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()`, `exclp()`, `execlepe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnvp()`, `spawnve()`, `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.
spawnlp()

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><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 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.
spawnlp()

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Read the 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>
spawnlp()  © 2006, QNX Software Systems GmbH & Co. KG.

Caveats:

If `mode` is P_WAIT, this function is a cancellation point.

See also:

`execl()`, `execle()`, `execlpe()`, `execv()`, `execve()`, `execvp()`,
`execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`,
`spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`,
`wait()`, `waitpid()`
Spawnlpe()  

Spawn a child process, given a list of arguments, an environment, and a relative path.

Synopsis:

```
#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.
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.

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,
spawnlpe()

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>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 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. |
spawnlpe()

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 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
spawnlpe()

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()`, `execlp()`, `execlpe()`, `execv()`, `execve()`, `execvp()`, `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`, `wait()`, `waitpid()`
**spawnp()**  
© 2006, QNX Software Systems GmbH & Co. KG.

Create and execute a new child process, given a relative path

**Synopsis:**

```c
#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()`. 

---

3188   C Library — S   November 2, 2006
If you want to `spawnp()` remotely, set the `nd` member of the inheritance structure to the node descriptor. See the `netmgr_strtond()` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argv</code></td>
<td>A pointer to an argument vector. The value in <code>argv[0]</code> should represent the filename of the program being loaded, but can be NULL if no arguments are being passed. The last member of <code>argv</code> must be a NULL pointer. The value of <code>argv</code> can’t be NULL.</td>
</tr>
<tr>
<td><code>envp</code></td>
<td>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: <code>variable=value</code> that’s used to define an environment variable. If the value of <code>envp</code> is NULL, then the child process inherits the environment of the parent process.</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 `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 `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>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 <code>spawn()</code>.</td>
</tr>
<tr>
<td>v</td>
<td>A vector of arguments to the program.</td>
</tr>
</tbody>
</table>

For a diagram of how the `spawn*` functions are related, see the
description of `spawn()`.

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_stime, 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()`.

The following arguments are passed to the underlying call (spawn()):

```c
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 `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 `inheritence` structure, `spawnp()` 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.
.spawnp()

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 or file’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.

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 spawnp() 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.
spawnp()

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>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`exec()`, `execl()`, `exect()`, `exectpe()`, `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()

Spawn 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:
<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 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.
- **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 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 };

     ;
    spawnv( P_WAIT, "myprog", arg_list );

    The program is found if myprog is in the current working directory.

Classification:

    QNX 4
**spawnv()**

**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:**

- `exec()`, `execle()`, `execl()`, `execlp()`, `execvp()`, `execve()`, `execvp()`, `execve()`, `execvpe()`, `execve()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnvp()`, `spawnve()`, `spawnv()`, `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:

```
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 `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()`):
spawnve()

spawnve(mode, path, argv, envp)

- 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 spawn(path, 0, 0, attr, argv, envp).

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 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 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.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Insufficient resources available to create the child process.</td>
</tr>
<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

<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(),
execve(), getenv(), putenv(), setenv(), spawn(), spawnl(), spawnle(),
spawnlpe(), spawnlp(), spawnp(), spawnvp(), spawnvpe(),
wait(), waitpid()
Synopsis:

```
#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.
**spawnvp()**

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 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><strong>mode</strong></th>
<th><strong>Return value</strong></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</td>
</tr>
<tr>
<td></td>
<td><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</td>
</tr>
<tr>
<td></td>
<td>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.
spawnvp()

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(), execlpe(), execlepe(), execv(), execve(), execvp(), execvepe(), getenv(), putenv(), setenv(), spawn(), spawnl(), spawnle(), spawnlpe(), spawnlpe(), spawn(), spawnv(), spawnve(), spawnvp(), spawnvpe(), wait(), waitpid()
spawnvpe()

Spawn a child process, given a vector of arguments, an environment, and a relative path

Synopsis:

```c
#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.
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(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.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 file doesn’t have the execute bit set.</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>Insufficient resources available to create the child process.</td>
</tr>
<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 file plus its 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 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 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 spawnvpe() function isn’t implemented for the filesystem specified in file.</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
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

**Caveats:**

If `mode` is P_WAIT, this function is a cancellation point.

**See also:**

- `execl()`, `execlp()`, `exclpe()`, `execv()`, `execve()`, `execvp()`,
- `execvpe()`, `getenv()`, `putenv()`, `setenv()`, `spawn()`, `spawnl()`, `spawnle()`,
- `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`,
- `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(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
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( "%.f\n", sqrt(.5) );
}
```
return EXIT_SUCCESS;
}

produces the output:

0.707107

**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, 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 ) {
```
```c
printf( "%d\n", rand() );
}

/*
   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()
Synopsis:

```
#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
See also:

`drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `mrand48()`, `nrand48()`, `seed48()`
**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 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.

After initialization, a state array can be restarted at a different point in one of two ways:

- The initstate() function can be used, with the desired seed, state array, and size of the array.
- The setstate() function, with the desired state, can be used, followed by 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.

**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>

**See also:**

drand48(), initstate(), rand(), random(), setstate(), srand()
Allocate, reallocate or free a block of memory

**Synopsis:**

```c
#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 `ptr` if there is a large enough block of unallocated memory immediately following `ptr`.

  Or:

- Allocating a new block, and copying the contents of `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 `ptr` can’t be reallocated. In this case, the memory pointed to by `ptr` isn’t freed, so be sure to keep a pointer to the old memory block.

```c
buffer = (char *) _srealloc( buffer, 100, 200 );
```

In the above example, `buffer` is set to NULL if the function fails, and no longer points to the old memory block. If `buffer` is your only pointer to the memory block, then you’ve lost access to this memory.

The `_srealloc()` function reallocates memory from the heap.

You must use `_sfree()` to deallocate the memory allocated by `_srealloc()`.

**Returns:**

A pointer to the start of the reallocated memory, or NULL if there’s insufficient memory available, or if the value of the `new_size` argument is zero.

**Classification:**

QNX Neutrino
**_srealloc()**

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

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

See also:

`fscanf()`, `fscanf()`, `scanf()`, `swscanf()`, `vfscanf()`, `vfwscanf()`, `vscanf()`, `vsscanf()`, `vswscanf()`, `vwscanf()`, `wscanf()`
Get information about a file or directory, given a path

Synopsis:

```c
#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>`:
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__)
            ino_t st_ino;
            ino_t st_ino_hi;
            off_t st_size;
            off_t st_size_hi;
        #elif defined(__BIG_ENDIAN__)
            ino_t st_ino_hi;
            ino_t st_ino;
            off_t st_size;
            off_t st_size_hi;
        #else
            #error endian not configured for system
        #endif
    #else
        #error _FILE_OFFSET_BITS value is unsupported
    #endif
    dev_t st_dev;   /* ID of the device containing the file. */
    dev_t st_rdev;  /* Device ID. */
    uid_t st_uid;   /* User ID of file. */
    gid_t st_gid;   /* Group ID of file. */
    time_t st_mtime; /* Time of last data modification. */
    time_t st_atime; /* Time when file data was last accessed. */
    time_t st_ctime; /* Time of last file status change. */
    mode_t st_mode; /* File types and permissions. */
    nlink_t st_nlink; /* Number of hard links to the file. */
    blksize_t st_blksize; /* Size of a block used by st_nblocks. */
    _int32_t st_blocks; /* Number of blocks st_blksize blocks. */
    blksize_t st_blksize; /* Preferred I/O block size for object. */
    #if _FILE_OFFSET_BITS - 0 == 64
        blkcnt_t st_blocks; /* No. of 512-byte blocks allocated for a file. */
    #elif defined(_FILE_OFFSET_BITS) && _FILE_OFFSET_BITS == 32
        #if defined(__LITTLE_ENDIAN__)
            blkcnt_t st_blocks;
        #elif defined(__BIG_ENDIAN__)
            blkcnt_t st_blocks_hi;
        #else
            #error endian not configured for system
        #endif
    #else
        #error _FILE_OFFSET_BITS value is unsupported
    #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_IRGRP</td>
<td>S_IROTH</td>
<td>Read</td>
</tr>
<tr>
<td>S_IRWXU</td>
<td>S_IRWXG</td>
<td>S_IRWXO</td>
<td>Read, write, execute/search. A bitwise inclusive OR of the other three constants. (S_IRWXU is OR of S_IRUSR, S_IWUSR 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:

<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 *st_mode* are defined for these file types:

- `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 following macros test whether a file is of a specified type. The value *m* supplied to the macros is the value of the *st_mode* field of a `stat` structure. The macros evaluate to a nonzero value if the test is true, and zero if the test is false.

- `S_ISBLK(m)` Test for block special file.
- `S_ISCHR(m)` Test for character special file.
- `S_ISDIR(m)` Test for directory file.
- `S_ISIFO(m)` Test for FIFO.
- `S_ISLNK(m)` Test for symbolic link.
- `S_ISNAM(m)` Test for special named file.
- `S_ISREG(m)` Test for regular file.
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 `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 `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(device)`
Extract the major number from a device ID.

`minor(device)`
Extract the minor number from a device ID.

`makedev(node, major, minor)`
Build a device ID from the given numbers. Currently, the `node` argument isn’t used and must be zero.

The `st_rdev` member of the `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>
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:

#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

<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`, `fstat()`, `fstat()`, `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 fbsize`
  The preferred filesystem blocksize.
\textbf{unsigned long} \textit{f_frsize}  \\
\hspace{1em} The fundamental filesystem blocksize (if supported)

\textbf{fsblkcnt_t} \textit{f_blocks}  \\
\hspace{1em} The total number of blocks on the filesystem, in units of \textit{f_frsize}.

\textbf{fsblkcnt_t} \textit{f_bfree}  \\
\hspace{1em} The total number of free blocks.

\textbf{fsblkcnt_t} \textit{f_bavail}  \\
\hspace{1em} The number of free blocks available to a nonsuperuser.

\textbf{fsfilcnt_t} \textit{f_files}  \\
\hspace{1em} The total number of file nodes (inodes).

\textbf{fsfilcnt_t} \textit{f_ffree}  \\
\hspace{1em} The total number of free file nodes.

\textbf{fsfilcnt_t} \textit{f_favail}  \\
\hspace{1em} The number of inodes available to a nonsuperuser.

\textbf{unsigned long} \textit{f_fsid}  \\
\hspace{1em} The filesystem ID (dev for now).

\textbf{char} \textit{f_basetype}[16]  \\
\hspace{1em} The type of the target filesystem, as a null-terminated string.

\textbf{unsigned long} \textit{f_flag}  \\
\hspace{1em} A bitmask of flags; the function can set these flags:
\begin{itemize}
  \item ST_RDONLY — read-only filesystem.
  \item ST_NOSUID — the filesystem doesn’t support \texttt{setuid/setgid} semantics.
\end{itemize}

\textbf{unsigned long} \textit{f_namemax}  \\
\hspace{1em} The maximum filename length.
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

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

Caveats:

The values returned for `files`, `ffree`, and `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()`
**stderr**

The standard error stream

**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:

```
#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`
Synopsis:

```c
#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:

```c
#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 ) { 
```
```c
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
### strcasecmp()

**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:

- strcmp()
- strcmpi()
- strcoll()
- stricmp()
- strncasecmp()
- strncmp()
- strnicmp()
- wcscmp()
- wcscoll()
- wcsncmp()
**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 );
}
```
```c
    return EXIT_SUCCESS;
}
```

produces the output:

```
Hello world
```

**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:**

`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

Cancellation point No
Interrupt handler Yes
Signal handler Yes
Thread Yes

See also:

`memchr()` `strcspn()` `strpbrk()` `strrchr()` `strspn()` `strstr()` `strtok()` `strtok_r()` `wcschr()` `wcsstr()` `wcspbrk()` `wcsrchr()` `wcsspn()` `wcsstr()` `wcstok()`
**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" ) );
}```
```c
    printf("%d\n", strcmp("mnopqr", "abcdef");
    return EXIT_SUCCESS;
}
```

produces the output:

```
0
1
-1
-1
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>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()`, `strcmpi()`, `strcoll()`, `stricmp()`, `strncasecmp()`, `strncmp()`,
`strnicmp()`, `wcscmp()`, `wscoll()`, `wcsncmp()`
**strcmpi()**

Compare two strings, ignoring case

**Synopsis:**

```c
#include <string.h>

int strcmpi( 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 `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>
#include <string.h>

int main( void )
```

{
    printf( "%d\n", strcmpi( "AbCDEF", "abcdef" ) );
    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

<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(), strcoll(), stricmp(), strncasecmp(), strncmp(), strnicmp(), wcscmp(), wcscoll(), wcscmp()
strcoll()  © 2006, QNX Software Systems GmbH & Co. KG.

Compare two strings, using the locale’s collating sequence

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 )
```
strcoll()

{  
    if (strcoll(buffer, "Hello") < 0) {  
        printf("Less than\n");  
    }  

    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:

setlocale(), strcasecmp(), strcmp(), strncmp(), stricmp(), strnecmp(), strncmp(), strnicmp(), wcsncmp(), wcscmp(), wcscoll(), wcsncmp()
Synopsis:

```c
#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:

```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>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(), wcscncpy(), wmemmove()
strcspn()  © 2006, QNX Software Systems GmbH & Co. KG.
Count the characters at the beginning of a string that aren’t in a given character set

Synopsis:

```
#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 NUL character isn’t considered part of `str`.

Returns:

The length of the initial segment.

Examples:

```
#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;
}

produces the output:

0
3
9

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(), strpbrk(), strrchr(), strspn(), strstr(), strtok(), strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcsrcr(), wcsspnn(), wcstr(), wcstok()
Create a duplicate of a string

Synopsis:

```c
#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:

```c
#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

<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:

free(), malloc(), memmove(), strcpy(), strncpy(), wcscpy(), wcscpy(), 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 ) {
```
```c
    printf( "Unable to open file: %s\n", 
            strerror( errno ) );
}

    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, perror(), stderr
**strftime()**

Format a time into a string

**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).

%R 24-hour notation; %H: %M.

%S Second as a decimal number (00-59).

%\t Tab character.

%T 24-hour clock time in the format $HH:MM:SS$.

%u Weekday as a decimal number (1-7), where Monday is 1.

%U Week number of the year as a decimal number (00-52), where Sunday is the first day of the week.

%V Week number of the year as a decimal number (01-53), where Monday is the first day of the week. The week containing January 1 is in the new year if four or more days are in the new year, otherwise it is the last week of the previous year.

%w Weekday as a decimal number (0-6), where 0 is Sunday.

%W Week number of the year as a decimal number (00-52), where Monday is the first day of the week.

%x Locale’s appropriate date representation.

%x Locale’s appropriate time representation.

%y Year without century, as a decimal number (00-99).

%Y Year with century, as a decimal number.

%z Offset from UTC -0430 (4 hrs, 30 minutes behind UTC, west of Greenwich), or no characters if time zone isn’t specified.

%Z Time zone name, or no characters if time zone isn’t specified.

%% Character %.

Some of the above conversion specifiers can be modified with the prefix E or O. If alternative formats don’t exist for the locale, they behave as if the unmodified conversion specifiers were called:
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%Ec</code></td>
<td>Alternative date and time representation.</td>
</tr>
<tr>
<td><code>%EC</code></td>
<td>Alternative name of the base year (period).</td>
</tr>
<tr>
<td><code>%Ex</code></td>
<td>Alternative date representation.</td>
</tr>
<tr>
<td><code>%EX</code></td>
<td>Alternative time representation.</td>
</tr>
<tr>
<td><code>%Ey</code></td>
<td>Offset from <code>%EC</code> of the alternative year (only) representation.</td>
</tr>
<tr>
<td><code>%EY</code></td>
<td>Alternative year representation.</td>
</tr>
<tr>
<td><code>%Od</code></td>
<td>Day of the month using alternative numeric symbols. Leading zeros are added if an alternative symbol for zero exists, otherwise leading spaces are used.</td>
</tr>
<tr>
<td><code>%Oe</code></td>
<td>Day of the month using alternative numeric symbols. Leading spaces are used.</td>
</tr>
<tr>
<td><code>%OH</code></td>
<td>24-hour clock using alternative numeric symbols.</td>
</tr>
<tr>
<td><code>%OI</code></td>
<td>12-hour clock using alternative numeric symbols.</td>
</tr>
<tr>
<td><code>%Om</code></td>
<td>Month using alternative numeric symbols.</td>
</tr>
<tr>
<td><code>%OM</code></td>
<td>Minutes using alternative numeric symbols.</td>
</tr>
<tr>
<td><code>%OS</code></td>
<td>Seconds using alternative numeric symbols.</td>
</tr>
<tr>
<td><code>%Ou</code></td>
<td>Alternative week day number representation (Monday=1).</td>
</tr>
<tr>
<td><code>%OU</code></td>
<td>Alternative week day number representation (Rules correspond with <code>%U</code>).</td>
</tr>
<tr>
<td><code>%OV</code></td>
<td>Alternative week number representation. (Rules correspond with <code>%V</code>).</td>
</tr>
<tr>
<td><code>%Ow</code></td>
<td>Weekday as a number using alternative numeric symbols (Sunday=0).</td>
</tr>
<tr>
<td><code>%OW</code></td>
<td>Week number of the year using alternative numeric symbols (Monday is the first day of the week).</td>
</tr>
<tr>
<td><code>%Oy</code></td>
<td>Year offset from <code>%C</code> using alternative numeric symbols.</td>
</tr>
</tbody>
</table>
>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:

```
#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>
</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>

3272   C Library — S       November 2, 2006
See also:

`asctime()`, `asctime_r()`, `ctime()`, `ctime_r()`, `strftime()`, `tm`, `tzset()`, `wcsftime()`
**stricmp()**

Compare two strings, ignoring case

**Synopsis:**

```c
#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:**

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

int main( void )
{
    printf( "%d\n", stricmp( "AbCDEF", "abcdef" ) );
}
```
```c
printf( "%d\n", stricmp( "abcdef", "ABC" ) );
printf( "%d\n", stricmp( "abc", "ABCdef" ) );
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

<table>
<thead>
<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()`, `strnca mecmp()`, `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:
strlwr()

A mixed-case STRING
a mixed-case string
a mixed-case 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>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()
strncasecmp()  
Compare two strings, ignoring case, up to a given length

Synopsis:
#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  sl is less than s2.
0  sl is equal to s2.
> 0  sl is greater than s2.

Examples:
The following code:

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

C Library — S
November 2, 2006
#include <stdlib.h>

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
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(), strcmpeq(), strcoll(), stricmp(), strncmp(), strnicmp(), wcscmp(), wcsncmp(), wcsncmp()
**Synopsis:**

```c
#include <string.h>

char* strncat( char* dst,
               const char* src,
               size_t n );
```

**Arguments:**

- `dst` and `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 )
{
```
strncpy( buffer, "Hello " );
strcpy( 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

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:

strcat()
**Synopsis:**
```c
#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:**
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdlib.h>

int main( void )
```
```c
{ 
    printf( "%d\n", strncmp( "abcdef", "abcDEF", 10 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF",  6 ) );
    printf( "%d\n", strncmp( "abcdef", "abcDEF",  3 ) );
    printf( "%d\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:**

strcasecmp(), strcmp(), strcmpi(), strcoll(), stricmp(), strncasecmp(), strnicmp(), wcscmp(), wcscoll(), wcscmp()
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.
strncpy()  

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

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:

`memmove()`, `strcpy()`, `strdup()`, `wcscpy()`, `wcsncpy()`, `wmemmove()`
strnicmp()  Compare two strings up to a given length, ignoring case

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 Library — S  November 2, 2006
strnicmp() function produces the output:
-20
-20

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(), strcmpl(), strcoll(), stricmp(), strncaasecmp(), strncmp(), wcscmp(), wcscoll(), wcscmp()
strnset() © 2006, QNX Software Systems GmbH & Co. KG.
Fill a string with a given character, to a given length

Synopsis:
#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:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char source[] = { "A sample STRING" };
int main( void )
{
    printf( "%s\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

<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:

strset()
strpbrk()  © 2006, QNX Software Systems GmbH & Co. KG.

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

Synopsis:

#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:

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:

#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" );
    }
}
 produces the output:

Find all vowels
ind all vowels
all vowels
owels
owels
els

Classification:

ANSI, POSIX 1003.1

Safety

Cancellation point  No
Interrupt handler   Yes
Signal handler     Yes
Thread             Yes

See also:

`memchr()`, `strchr()`, `strcspn()`, `strstr()`, `strspn()`, `strtok()`,
`strtok_r()`, `wcschr()`, `wcespn()`, `wcespbrk()`, `wcsrrchr()`, `wcsspbn()`,
`wcstr()`, `wcstok()`
strrchr() © 2006, QNX Software Systems GmbH & Co. KG.

Find the last occurrence of a character in a string

Synopsis:

```c
#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:

```c
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:

```c
#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;
}
```
produces the output:

```
abcde
NULL
```

**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:**

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcespbrk()`, `wcesrchr()`, `wcsstr()`, `wcstok()`
**strrev()**

Reverse a string

**Synopsis:**

```
#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:**

```
#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>
**strsep()**

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

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:

strtok(), strtok_r(), wcstok()
**strset()**

Fill a string with a given character

**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, '*' ) );
}
```
```c
return EXIT_SUCCESS;
}
```

produces the output:

```
A sample STRING
***************
```

**Classification:**

QNX 4

**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:**

`strnset()`
strsignal()

Return the description of a signal

Synopsis:

```c
#include <string.h>

char *strsignal( int signo );
```

Arguments:

`signo` The signal number that you want the description of.

Library:

```c
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()  

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Yes/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:

setlocale()
strspn()  © 2006, QNX Software Systems GmbH & Co. KG.

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( "\n", strspn( "out to lunch", "aeiou" ) );
    printf( "\n", strspn( "out to lunch", "xyz" ) );
    return EXIT_SUCCESS;
}
```
strspn() produces the output:

2
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:

memchr(), strchr(), strcspn(), strpbrk(), strrchr(), strstr(), strtok(), strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcsrchr(), wcsspn(), 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:
**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()`, `wcspbrk()`, `wcsrchr()`, `wcsspn()`, `wcsstr()`, `wcstok()`
**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:**

- `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\n",pi );
    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:

`atof()`, `errno`
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:

```c
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()`, `strtoul()`, 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 | 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

<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()`, `strtoul()`
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`
strtok()

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>No</td>
</tr>
</tbody>
</table>

See also:

memchr(), strchr(), strcspn(), strpbrk(), strrchr(), strset(), strspn(),
strstr(), strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcsrchr(),
wcsspn(), wcsstr(), wcstok()
strtok_r()  
Break a string into tokens (reentrant)

Synopsis:

```c
#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 `sep` may be different from call to call. When no tokens remain in `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

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:

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strset()`, `strspn()`, `strstr()`, `strtok()`, `wcschr()`, `wcscspn()`, `wcspbrk()`, `wcsrchr()`, `wcspan()`, `wcsstr()`, `wcstok()`
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` or `LONG_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:

```
#include <stdlib.h>

int main( void )
{
    long int v;

    v = strtol( "12345678", NULL, 10 );
    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:**

`atoi()`, `atol()`, `errno`, `itoa()`, `ltoa()`, `sscanf()`, `strtoul()`, `ultoa()`, `utoa()`
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 \texttt{-l c} option to \texttt{qcc} to link against this library. This library is usually included automatically.

Description:

The \texttt{strtol()} function converts the string pointed to by \texttt{ptr} to an \texttt{unsigned long}; \texttt{stroull()} converts the string pointed to by \texttt{ptr} to an \texttt{unsigned long long}.

These functions recognize strings that contain the following:

\begin{itemize}
  \item optional white space
  \item a sequence of digits and letters.
\end{itemize}

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.

Returns:

The converted value.

If the correct value causes an overflow, \texttt{ULONG_MAX} or \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}.

Examples:

\begin{verbatim}
#include <stdlib.h>

int main( void )
{
  unsigned long int v;

  v = strtol( "12345678", NULL, 10 );
  return EXIT_SUCCESS;
}
\end{verbatim}
Classification:

ANSI, POSIX 1003.1

Safety

<table>
<thead>
<tr>
<th></th>
<th>Ans</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

<table>
<thead>
<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:

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:**

```c
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 *strncpy()* 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**

- Cancellation point: No
- Interrupt handler: Yes

continued…
strxfrm()

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:

setlocale(), strcoll(), wcxstrm()
**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-1</code> 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

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:

`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 `printf()`.
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

<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(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), 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

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, fscanf(), fwscanf(), scanf(), sscanf(), vfscanf(), vfwscanf(), vscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()
symlink()
Create a symbolic link to a path

Synopsis:

```c
#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 then `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.

EROF S The file `slink` would reside on a read-only filesystem.

Examples:

```c
/*
 * 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()

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

<table>
<thead>
<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:

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 `PTHREADCOND_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()

<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()
SyncCondvarWait(), SyncCondvarWait_r()

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 \texttt{STATE\_MUTEX} state waiting for \textit{mutex} to be unlocked. At this point, it’s as though you had called \texttt{SyncMutexLock(mutex)}.

In all cases, \textit{mutex} is reacquired before the call returns. If the thread enters the \texttt{STATE\_MUTEX} state, the rules governing \texttt{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**

\texttt{STATE\_CONDVAR}

The calling thread blocks waiting for the condition variable to be signaled.

\texttt{STATE\_MUTEX}

The thread was unblocked from the \texttt{STATE\_CONDVAR} state and while trying to reacquire the controlling \textit{mutex}, found the \textit{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:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>On the first use of a statically initialized sync, all kernel synchronization objects were in use.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>A fault occurred when the kernel tried to access sync or mutex.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The synchronization ID specified in sync doesn’t exist.</td>
</tr>
<tr>
<td>ETIMEDOUT</td>
<td>A kernel timeout unblocked the call. See TimerTimeout().</td>
</tr>
</tbody>
</table>

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:

pthread_cond_broadcast(), pthread_cond_signal(),
pthread_cond_timedwait(), pthread_cond_wait(),
pthread_mutex_lock(), SignalKill(), SyncCondvarSignal(),
SyncMutexLock(), SyncTypeCreate(), ThreadCancel(),
TimerTimeout()
Synopsis:

```
#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:

```
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.
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:

`pthread_mutex_getprioceiling()`, `pthread_mutex_setprioceiling()`, `SyncCondvarSignal()`, `SyncCondvarWait()`, `SyncDestroy()`, `SyncMutexEvent()`, `SyncMutexLock()`, `SyncMutexRevive()`, `SyncMutexUnlock()`, `SyncTypeCreate()`
**SyncDestroy(), SyncDestroy_r()** © 2006, QNX Software Systems GmbH & Co. KG.

Destroy a synchronization object

**Synopsis:**

```c
#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()`).

☞ 3352  C Library — S November 2, 2006
Blocking states

These calls don’t block.

Returns:

The only difference between these functions is the way they indicate errors:

\textit{SyncDestroy()}

If an error occurs, the function returns -1 and sets \textit{errno}. Any other value returned indicates success.

\textit{SyncDestroy_r()}

Returns EOK on success. This function does NOT set \textit{errno}. If an error occurs, the function can return any value listed in the Errors section.

Errors:

\begin{itemize}
  \item \textbf{EBUSY} The synchronization object is locked by a thread.
  \item \textbf{EFAULT} A fault occurred when the kernel tried to access \textit{sync}.
  \item \textbf{EINVAL} The synchronization ID specified in \textit{sync} doesn’t exist.
\end{itemize}

Classification:

QNX Neutrino

\begin{tabular}{|c|c|}
  \hline
  \textbf{Safety} & \  \\
  \textbf{Cancellation point} & No \\
  \textbf{Interrupt handler} & No \\
  \textbf{Signal handler} & Yes \\
  \textbf{Thread} & Yes \\
  \hline
\end{tabular}
SyncDestroy(), SyncDestroy_r() © 2006, QNX Software Systems GmbH & Co. KG.

See also:

pthread_cond_destroy(), pthread_mutex_destroy(),
pthread_rwlock_destroy(), sem_destroy(), SyncTypeCreate()
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.
SyncMutexEvent(), SyncMutexEvent_r() © 2006, QNX Software Systems GmbH & Co. KG.

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

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>

3356 C Library — S November 2, 2006
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.

Par
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(), 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

<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_unlock(), SyncTypeCreate(), SyncDestroy(), SyncMutexUnlock()
**SyncMutexRevive(), SyncMutexRevive_r()**

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(), 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

| Cancellation point | No |
| Interrupt handler  | No |
| Signal handler     | Yes|
| Thread             | Yes|

See also:

pthread_mutex_lock(), pthread_mutex_unlock(), SyncTypeCreate(), SyncDestroy(), SyncMutexEvent(), SyncMutexLock(), SyncMutexUnlock()
SyncMutexUnlock(),
SyncMutexUnlock_r()

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

<table>
<thead>
<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_lock()`, `pthread_mutex_unlock()`, `SyncTypeCreate()`, `SyncMutexLock()`
**SyncSemPost(), SyncSemPost_r()**

GmbH & Co. KG.

Increment a semaphore

**Synopsis:**

```c
#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_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

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:

sem_destroy(), sem_init(), sem_post(), sem_trywait(), sem_wait(),
SyncDestroy(), SyncSemWait(), SyncTypeCreate()
SyncSemWait(), SyncSemWait_r() © 2006, QNX Software Systems

Wait on a semaphore

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>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>

November 2, 2006 C Library — S 3369
See also:

sem_destroy(), sem_init(), sem_post(), sem_timedwait(), sem_trywait(), sem_wait(), SyncDestroy(), SyncSemPost(), SyncTypeCreate()
**Synopsis:**

```c
#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_SYNC_SEM — 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 `-l 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…
SyncTypeCreate(),
SyncTypeCreate_r()

<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>count</td>
<td>&gt;0</td>
<td>Recursive counter number of the mutex</td>
</tr>
<tr>
<td>count</td>
<td>&lt;1</td>
<td>If the high bit of count 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:

EAGAIN     All kernel synchronization objects are in use.
EFAULT     A fault occurred when the kernel tried to access sync or attr.
EINVAL     Either
- 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>
</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_mutex_init(), pthread_rwlock_init(), sem_init(), SyncCondvarSignal(), SyncCondvarWait(), SyncDestroy(), SyncMutexLock(), SyncMutexUnlock()
sysconf()

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 `<confname.h>`, 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()

 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
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:

```c
#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) stylesheet 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 `name_len` 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

<sys/socket.h> Second-level network identifiers
<netinet/in.h> Third-level Internet identifiers and fourth-level IP identifiers
<netinet/icmp_var.h> Fourth-level ICMP identifiers
<netinet/tcp_var.h> Fourth-level TCP identifiers
<netinet/udp_var.h> Fourth-level UDP identifiers

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:

<table>
<thead>
<tr>
<th>Protocol name</th>
<th>Variable name</th>
<th>Type</th>
<th>Changeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>forwarding</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>redirect</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>ttl</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>forwsrct</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>directed-broadcast</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>allowsrct</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>subnetsarelocal</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>mtudisc</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>maxfragpackets</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>sourcecheck</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>sourcecheck_logint</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td>icmp</td>
<td>maskrepl</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td>tcp</td>
<td>rfc1323</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>sendspace</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>recvspace</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>mssdflt</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>syn_cache_limit</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>syn_bucket_limit</td>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>syn_cache_interval</td>
<td>Integer</td>
<td>Yes</td>
</tr>
</tbody>
</table>

continued...
The variables are as follows:

**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.
**sysctl()**

- **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.
sysctl()

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

Safety

- Cancellation point: Yes
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

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:
#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>
</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:

procnto in the Utilities Reference
SYSPAGE_CPU_ENTRY()
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 `-l c` 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.
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

<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:

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()

Return an entry from the system page

Synopsis:

#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().
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( );

    /* do something */
    printf("poo\n");

    /* 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

continued...
**SYSPAGE_ENTRY()**

**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:**

`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:

```c
#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()`
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

<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:

abort(), atexit(), close(), errno, execl(), execlp(), execle(), execvp(), execve(), execvp(), exit(), _exit(), getenv(), main(), putenv(), sigaction(), signal(), spawn(), spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnp(), spawnvp(), spawnvpe(), 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:**

```c
#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:**

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

int main( void )
```

---

© 2006, QNX Software Systems GmbH & Co. KG.
\begin{verbatim}
\{ 
   printf( "%f\n", tan(.5) );
   return EXIT_SUCCESS;
\}
\end{verbatim}

produces the output:

0.546302

**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:**

atan(), atan2(), cos(), sin()
Calculate the hyperbolic tangent

**Synopsis:**

```c
#include <math.h>

double tanh( double x );

float tanhf( float x );
```

**Arguments:**

- `x` The angle, in radians, for which you want to compute the hyperbolic tangent.

**Library:**

- `libm`

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

**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.

**Returns:**

The hyperbolic tangent 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 <stdlib.h>
#include <math.h>

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>
<th></th>
</tr>
</thead>
<tbody>
<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, sinh()
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:

- **EBADF** The argument `fildes` is invalid.
- **EINTR** A signal interrupted the operation.
- **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 );
    write( fildes, "ATH", 3 );

    /* Wait for data to transmit before returning */
    tcdrain( fildes );
    close( fildes );
    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:

`write()`
tcdropline()

Disconnect a communications line

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).
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()`
**tcflow()**

**Perform a flow-control operation on a data stream**

**Synopsis:**

```c
#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`.
- **TCOOFFHW** 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 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 fildes (that is, software flow control).

TCIONHW

Cause input to be resumed by using hardware flow control.

Returns:

0  Success.

-1  An error occurred (errno is set).

Errors:

EBADF  Invalid fildes argument.

EINVAL  Invalid action 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:

```
#include <termios.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
```
int fd;
fd = open("/dev/ser1", O_RDWR);
/* Resume output on flow-controlled device */
tcflow(fd, TCOON);
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:
tcdrain(), tcflush(), tcsendbreak()
Flush the input and/or output stream

Synopsis:

```c
#include <termios.h>

int tcflush( int fildes,  
            int queue_selector );
```

Arguments:

- `fildes` - A file descriptor that's associated with the data stream that you want to perform the operation on.
- `queue_selector` - The stream or streams that you want to flush. At least the following values for `queue_selector` are defined in `<termios.h>`:
  - TCIFLUSH — discard all data that's received, but not yet read, on the device associated with `fildes`.
  - TCOFLUSH — discard all data that's written, but not yet transmitted, on the device associated with `fildes`.
  - 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 `fildes`.

Library:

- `libc`

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

Description:

The `tcflush()` function flushes the input stream, the output stream, or both, depending on the value of the argument `queue_selector`. 
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, TCIFLUSH );

    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()
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>
</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, fpathconf(), tcsetattr(), termios

Chapter 7 of POSIX 1003.1
tcgetpgrp()
Get the process group ID associated with a device

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.
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()`
tcgetsid()

Get the process group ID of the session leader for a controlling terminal

Synopsis:

```c
#include <unistd.h>

int tcgetsid( int filedes );
```

Arguments:

`fdedes` 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:

- `EACCESS` 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:

```
#include <termios.h>

int tcgetsize( int filedes,
               int* prow,
               int* pcols );
```

Arguments:

- `filedes` 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` is a file descriptor that’s associated with the device whose input buffer you want to add characters to.
- `buf` is a pointer to a buffer that contains the characters that you want to insert.
- `n` is 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:

```c
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

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…
## tcinject()

### Safety

<table>
<thead>
<tr>
<th>Thread</th>
<th>Yes</th>
</tr>
</thead>
</table>

November 2, 2006
**tcischars()**

*© 2006, QNX Software Systems GmbH & Co. KG.*

**Determine the number of characters waiting to be read**

**Synopsis:**

```c
#include <termios.h>

int tcischars(int filedes);
```

**Arguments:**

- `filedes` 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, `filedes`.

**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

---

3430 C Library — T to Z

November 2, 2006
### 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>
Internet Transmission Control Protocol

Synopsis:

```c
#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(), getprotobynmae(), getsockopt(), listen(), setsockopt(), socket()

RFC 793
Synopsis:

```c
#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 *fd* is invalid.
- **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 );

    /* Send a 500 millisecond break */
    tcsendbreak( fd, 500 );

    close( fd );
    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:

tcflush(), 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()`.
Returns:

0   Success.

-1  An error occurred; errno is set.

Errors:

EBADF  The argument fildes is invalid;

EINVAL 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 main( void )
{
    raw( 0 );
    /*
     * Stdin is now "raw"
     */
    unraw( 0 );
    return EXIT_SUCCESS;
}

int raw( fd )
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( 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 ) );
}

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, select(), tcgetattr(), termios
Synopsis:

```c
#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).
Errors:

EBADF The argument \texttt{fildes} is invalid.
EINV AL The argument \texttt{pgrp id} is invalid.
ENOSYS The resource manager associated with \texttt{fildes} doesn’t support this call.
ENOTTY The argument \texttt{fildes} isn’t associated with a terminal device.
EPERM The argument \texttt{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

<table>
<thead>
<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:

signal(), tcgetpgrp()
 tcsetsid()  
Make a terminal device a controlling device

Synopsis:

```c
#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 c` 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, TIOCSCTTY)` 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.
EINVVAL   The argument \textit{pid} is invalid.
ENOSYS, ENOTTY    
                   The argument \textit{fd} isn’t associated with a terminal device.

Classification:

QNX Neutrino

\begin{tabular}{l|c}
\textbf{Safety} & \\
Cancellation point & No \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\end{tabular}

See also:

\textit{ioctl()}, \textit{tcgetsid()}

November 2, 2006  C Library — T to Z  3445
**Synopsis:**
```
#include <termios.h>

int tcsetsize( int filedes,
              int rows,
              int cols );
```

**Arguments:**
- `fd` : 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.
tcsetsize()

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:

tcgetsize()
**tell(), tell64()**

Determine the current file position

**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

<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()`, `creat()`, `dup()`, `dup2()`, `eof()`, `errno`, `execl()`, `execle()`, `execl()`, `execle()`, `execv()`, `execve()`, `execvp()`, `execve()`, `fcntl()`, `fileno()`, `fstat()`, `isatty()`, `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>Feature</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>No</td>
<td></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.

**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>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 `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()`
termios

© 2006, QNX Software Systems GmbH & Co. KG.

Terminal control structure

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.

**tcflag_t c_cflag**

Control modes. This member contains at least the following bits:

- **CLOCAL**  Ignore modem status lines.
- **CREAD**  Enable receiver.
- **CSIZE**  Number of data bits per character.
  - CS5  5 data bits.
  - CS6  6 data bits.
  - CS7  7 data bits.
  - CS8  8 data bits.
- **CSTOPB**  Two stop bits, else one.
- **HUPCL**  Hang up on last close.
- **IHFLOW**  Support input flow control using the hardware handshaking lines.
- **OHFLOW**  Support output flow control using the hardware handshaking lines.
- **PARENB**  Parity enable.
- **PARODD**  Odd parity, else even.
- **PARSTK**  Stick parity (mark parity if PARODD is set, else space parity).

**tcflag_t c_lflag**

Local modes. This member contains at least the following bits:

- **ECHO**  Enable echo.
- **ECHOE**  Echo ERASE as destructive backspace.
- **ECHOK**  Echo KILL as a line erase.
ECHONL  Echo ‘\n’, even if ECHO is off.
ICANON  Canonical input mode (line editing enabled).
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.
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() © 2006, QNX Software Systems GmbH & Co. KG.

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

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:

thread_pool_destroy(), thread_pool_create(), thread_pool_limits(),
thread_pool_start()
thread_pool_create()

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_HANDLE_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`.
The function that’s called when the worker thread exits, to free the context allocated with `context_alloc`.

attr

A pointer to a `pthread_attr_t()` function that’s passed to `pthread_create()`. The `pthread_attr_t()` 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:

```c
/* 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_io_funcs_t io_funcs;
static iofunc_attr_t attr;

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.
", argv[0]);
    return EXIT_FAILURE;
}

iofunc_func_init( _RESMGR_CONNECT_NFUNCS,
&connect_funcs,
_RESMGR_IO_NFUNCS, &io_funcs );
iofunc_attr_init( &attr, _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/null", _FTYPE_ANY, 0, &connect_funcs,
&io_func, &attr )) == -1) {
    fprintf( stderr,
"%s: Unable to attach name.
", argv[0] );
    return EXIT_FAILURE;
}

/* Start the thread which will handle interrupt events. */
thread_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 Resource Manager chapter in the *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>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:**

0 Success.

-1 An error occurred.

**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>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>
thread_pool_destroy()

See also:

thread_pool_control(), thread_pool_create(), thread_pool_limits(), thread_pool_start()
**thread_pool_limits()**

*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></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:

thread_pool_control(), thread_pool_create(), thread_pool_destroy(), thread_pool_start()
**thread_pool_start()**

Start a thread pool

**Synopsis:**

```c
#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:**

```c
#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 ) {
    fprintf( stderr, "%s: Unable to allocate 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 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

continued…
thread_pool_start()

Safety

| Thread | Yes |

See also:

thread_pool_create(), thread_pool_destroy()
ThreadCancel(),
ThreadCancel_r()

Cancel a thread

Synopsis:

```
#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 `cansstub`. 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
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**

  Cancellation point  No

  continued…
ThreadCancel(), ThreadCancel_r() © 2006, 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>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()

Create a thread

Synopsis:

```c
#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.
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_NORETURN.

**struct sched_param** param

A sched_param structure that specifies the scheduling parameters, as defined by the SchedSet()
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.
ThreadCreate(), ThreadCreate_r()

PTHREAD_CANCEL_DEFERRED (default)

Cancellation occurs only at cancellation points as defined by ThreadCancel().

PTHREAD_CANCELASYCHRONOUS

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()** © 2006, QNX Software Systems GmbH & Co. KG.

- **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 the Errors section.

**Errors:**

- **EAGAIN**: All kernel thread objects are in use.
- **EFAULT**: A fault occurred when the kernel tried to access the buffers provided.
- **EINVAL**: Invalid scheduling policy or priority specified.
- **ENOTSUP**: PTHREAD_SCOPE_PROCESS was requested. All kernel threads are PTHREAD_SCOPE_SYSTEM.
**ThreadCreate(), ThreadCreate_r()**

EPERM  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

<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 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()`
**ThreadCtl(), ThreadCtl_r()**

Control a thread

**Synopsis:**

```c
#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.

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.
ThreadCtl(), ThreadCtl_r() © 2006, QNX Software Systems GmbH & Co. KG.

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.

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.

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, &(_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.
**ThreadCtl(), ThreadCtl_r()** © 2006, QNX Software Systems GmbH & Co. KG.

This call was defined in the QNX Neutrino Core OS 6.3.2.

*ThreadCtl( , , 0)*)

Unfreeze all threads in the current process that were frozen using the **_NTOTCTL_THREADS_HOLD** command.

*ThreadCtl( , , 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_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.

EINVAL The name buffer length is invalid or smaller than 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;

    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);
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>
</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:

`InterruptDisable()`, `InterruptEnable()`, `InterruptMask()`, `InterruptUnmask()`
ThreadDestroy(), ThreadDestroy_r() © 2006, QNX Software Systems GmbH & Co. KG.

Destroy a thread immediately

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 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.
ThreadDestroy(), ThreadDestroy_r()

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(), ThreadDestroy_r() © 2006, QNX Software Systems

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

<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_abort(), pthread_exit(), ThreadCancel(), ThreadCreate()
Synopsis:

```
#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.
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:

EINV AL The thread is already detached.
ESRCH The thread indicated by tid 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:

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.
```
ThreadJoin(), ThreadJoin_r()

EINTR       The call was interrupted by a signal.
EINV        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

<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>Yes</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
<td></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`.

© 2006, QNX Software Systems GmbH & Co. KG.
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
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
```

November 2, 2006
typedef union {
    struct pulse pulse;
    /* your other message structures would go here too */
} my_message_t;

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, chid, _NTO_SIDE_CHANNEL, 0);
    event.sigev_priority = getpid(0);
    event.sigev_code = MY_PULSE_CODE;
    timer_create(CLOCK_REALTIME, &event, &timer_id);

    itime.it_value.tv_usec = 1;
    /* 500 million nsecs = .5 secs */
    itime.it_value.tv_nsec = 500000000;
    itime.it_interval.tv_usec = 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_gettime()`, `timer_getexpstatus()`, `timer_getoverrun()`, `timer_settime()`
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

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes

### See also:

- `clock_getres()`, `clock_gettime()`, `clock_settime()`, `nanosleep()`, `sleep()`,
- `timer_create()`, `timer_getexpstatus()`, `timer_getoverrun()`,
- `timer_gettime()`, `timer_settime()`
timer_getexpstatus() © 2006, QNX Software Systems GmbH & Co. KG.

Get the expiry status of a timer

Synopsis:

```c
#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.
timer_getexpstatus()

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_create(), timer_delete(), timer_getovrun(), timer_gettime(),
timer_settime(), TimerInfo()
timer_getoverrun() — Return the number of timer overruns

Synopsis:

```
#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.
timer_getoverrun()

Returns:
The number of overruns, or -1 if an error occurs (errno is set).

Errors:
EINVAL 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 `struct itimerspec` structure that the function fills in with the timer’s time until expiry. The structure contains at least the following members:
  - `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 c` 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()  

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

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_delete(), timer_gettimestatus(),
timer_getoverrun(), timer_settime(), timespec
**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_gettime()`, `timer_getovrerrun()`, `timer_gettime()`
timer_timeout(),
timer_timeout_r()

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.
timer_timeout(), timer_timeout_r() © 2006, QNX Software Systems GmbH & Co. KG.

ntime A pointer to a timespec structure that specifies the timeout.

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 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:
### timer_timeout()

<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>
<tr>
<td>MsgSendv()</td>
<td>STATE_SEND or STATE_REPL Y</td>
</tr>
<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>

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()** © 2006, QNX Software Systems GmbH & Co. KG.

<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:

```
_NTO_TIMEOUT_SEND | _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,
               _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 `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`.
timer_timeout(), timer_timeout_r() © 2006, QNX Software Systems GmbH & Co. KG.

- Occurs on a clock tick (see ClockPeriod()) so the actual wakeup time is a minimum of:

  \[(tv_{sec} \times 1000000000 + tv_{nsec}) \div \text{(size of timer tick)}\] nanoseconds

  where \(tv_{sec}\) and \(tv_{nsec}\) are fields of the `timespec` structure (defined in <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 `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 `MsgReceivev()` with an immediate timeout. A much better approach is to use multiple threads and have one block waiting for messages.

If `flags` is set to `_NTO.TIMEOUT.NANOSLEEP`, then these calls block in the `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:

```c
timer_timeout( CLOCK_REALTIME, _NTO.TIMEOUT.NANOSLEEP,
                 NULL, &ntime, &otime );
```

If `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 `flags`. In this case, the calls block as follows:

**STATE.NANOSLEEP**

- The calling thread blocks for the requested time period.
timer_timeout()

The previous flags. If an error occurs, the function returns -1 and sets errno.

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

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 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()`
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.
    - 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.
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.
TimerAlarm(), TimerAlarm_r()

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()** © 2006, QNX Software Systems GmbH & Co.

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()`
**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>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</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` 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 `id`, and stores the information in the buffer pointed to by `info`.

The `TimerInfo()` and `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 `timer_gettime()`, `timer_getoverflow()`, or `timer_gettime()`.

```c
struct _timer_info
```

The `_timer_info` structure pointed to by `info` contains at least these members:

```c
uint32_t flags
```

One or more of these bit flags:

- `_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.

```c
int32_t tid
```

The thread to which the timer is directed (0 if it’s directed to the process).

```c
int32_t notify
```

The notify type.

```c
clockid_t clockid
```

The type of clock used.
uint32_t overruns
    The number of overruns.

struct sigevent event
    The event dispatched when the timer expires.

struct itimerspec itime
    Time when the timer was started.

struct itimerspec otime
    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:

EINVAL The timer specified by id doesn’t exist.

ESRCH The process specified by pid 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, 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.
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{TimerSettime()} and \texttt{timer_settime()} kernel calls set the expiration time of the timer specified by \textit{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 \texttt{timer_gettime()} or \texttt{timer_settime()}.

Expiration date

The expiration is specified by the \textit{itime} argument. The \texttt{.itime} structure contains at least the following members:

\begin{verbatim}
uint64_t nsec
  The expiration time to set.
uint64_t interval_nsec
  The interval reload time.
\end{verbatim}

If the \texttt{nsec} member of \textit{itime} is zero, then the timer is disarmed.

If the \texttt{interval_nsec} member of \textit{itime} is nonzero, then it specifies a repeat rate which is added to the timer once the \texttt{nsec} period has expired. Subsequently, the timer is automatically rearmed, causing it to become repetitive with a period of \texttt{interval_nsec}.

If the timer is already armed when you call \texttt{TimerSettime()}, this call discards the previous setting and sets a new setting.

If the event notification specified by \texttt{TimerCreate()} has a \texttt{SIGEV_CODE} of \texttt{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 oitime.

Classification:

QNX Neutrino

Safety

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

continued...
TimerSettime(), TimerSettime_r()

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:

timer_gettime(), timer_settime(), TimerCreate(), TimerInfo()
TimerTimeout(),
TimerTimeout_r()

Set a timeout on a blocking state

Synopsis:

#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 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()  © 2006, QNX Software Systems 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

Instead of using these kernel calls directly, consider calling

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>InterruptWait()</td>
<td>STATE_INTR</td>
</tr>
<tr>
<td>MsgReceivev()</td>
<td>STATE_RECEIVE</td>
</tr>
<tr>
<td>MsgSendv()</td>
<td>STATE_SEND or STATE_REPLY</td>
</tr>
<tr>
<td>SignalSuspend()</td>
<td>STATE_SIGSUSPEND</td>
</tr>
</tbody>
</table>

continued...
TimerTimeout(),
TimerTimeout_r()

<table>
<thead>
<tr>
<th>Call</th>
<th>Blocking state</th>
</tr>
</thead>
<tbody>
<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.
_NTO_TIMEOUT_SEND
  Timeout on STATE_SEND.
TimerTimeout(), TimerTimeout_r()

_GTO.TIMEOUT_SIGSUSPEND
Timeout on STATE_SIGSUSPEND.

_GTO.TIMEOUT_SIGWAITINFO
Timeout on STATE_SIGWAITINFO.

For example, to set a timeout on `MsgSendv()`, specify:

_GTO.TIMEOUT_SEND | _GTO.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:

```c
...
  event.sigev_notify = SIGEV.UNBLOCK;
  timeout = 10 * 1000000000;
  TimerTimeout( CLOCK_REALTIME, 
                 _GTO.TIMEOUT_SEND | _GTO.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 `MsgRepyv()`.

---

**The timeout**

The type of timer used to implement the timeout is specified with the `id` argument.

The timeout:

- Is specified by the `ntime` argument (the number of nanoseconds).
**TimerTimeout(), TimerTimeout_r()**

- Is relative to the current time (when `TimerTimeout()` 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:

  \[
  \frac{(ntime)}{\text{size of timer tick}} \text{ nanoseconds}
  \]

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 `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 `flags` immediately returns with ETIMEDOUT. Although a questionable practice, you can use it to poll potential blocking kernel calls. For example, you can poll for messages using `MsgReceivev()` with an immediate timeout. A much better approach is to use multiple threads and have one block waiting for messages.

If you set `flags` to _NTO_TIMEOUT_NANOSLEEP_, then these calls block in the 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:

```c
TimerTimeout( CLOCK_REALTIME, _NTO_TIMEOUT_NANOSLEEP,
               NULL, ntime, otime );
```

If `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 `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.
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.
EINTR The call was interrupted by a signal.
EINVAL Invalid timer value id.

Classification:

QNX Neutrino
TimerTimeout(), TimerTimeout_r()

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:

sigevent, timer_timeout(), TimerCreate(), TimerInfo()
Synopsis:

```
#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:

- `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.
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:

#define sysconf(_SC_CLK_TCK)

The times of a terminated child process are included in the tms_cstime 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:

/*
 * 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 );
}
times()

printf( "user time = %d\n", childtim.tms_cutime );
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:

clock_gettime()
Synopsis:

```c
#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>
See also:

nsec2timespec(), timespec
timezone
The number of seconds by which the local time zone is earlier than UTC

Synopsis:
```
#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
Structure that describes calendar time

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().

tm_zone  String for the time zone name.

Classification:

ANSI, POSIX 1003.1

See also:

asctime(), gmtime(), gmtime_r(), localtime(), localtime_r(), mktime(),
setlocale(), strftime(), wcsftime()
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).
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.

EROFSS  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

<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()
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_Jmpdir` (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`. 
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**

<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...
tmpnam()

Safety

Signal handler  No
Thread  Read the Caveats

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:

tmpnam(), tmpfile()
Synopsis:

```c
#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:

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

char chars[] = {
    'A',
    '5',
    '$',
    'Z'
};

#define SIZE sizeof( chars ) / sizeof( char )

int main( void )
{
```
```c
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()`
Synopsis:

```c
#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 a 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:

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

char chars[] = {
    'a',
    '5',
    '$',
    'z'
};

#define SIZE sizeof( chars ) / sizeof( char )

int main( void )
{
```
```c
int i;

for( i = 0; i < SIZE; i++ ) {
    printf( "%c", toupper( chars[ i ] ) );
}
printf( "\n" );
return EXIT_SUCCESS;
}
```

produces the output:

```plaintext
A 5 $ Z
```

**Classification:**

ANSI, POSIX 1003.1

**Safety**

- Cancellation point: No
- Interrupt handler: Yes
- Signal handler: Yes
- Thread: Yes

**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.
Errors:

EINV AL  The conversion descriptor in category is invalid.

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:

wctrans()

“Character manipulation functions” and “Wide-character functions” in Library Reference Summary
towlower()  
Convert a wide character to lowercase

Synopsis:

```c
#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>
</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>

3578  C Library — T to Z  November 2, 2006
See also:

“Character manipulation functions” and “Wide-character functions” in *Library Reference Summary*
towupper()  
Convert a wide character to uppercase

Synopsis:
```
#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>
See also:

“Character manipulation functions” and “Wide-character functions” in Library Reference Summary
TraceEvent()  Trace kernel events

Synopsis:

```c
#include <sys/neutrino.h>
#include <sys/trace.h>

int TraceEvent( int mode,
                ... );
```

Arguments:

- `mode`: A command that indicates what you want to trace. Certain modes require additional arguments.

Library:

- `libc`

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

Description:

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 internal circular link list of trace buffers
- initializing filters
- turning on or off the event stream
- deallocating the internal circular link list of trace buffers

This function requires the instrumented kernel. For more information, see the documentation for the System Analysis Toolkit (SAT).
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    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 the appropriate 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>
**TraceEvent()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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 <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.
**truncates**

© 2006, QNX Software Systems GmbH & Co. KG.

**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.

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.
truncatem() 

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:

#include <unistd.h>

char *ttyname(int fd);

Arguments:

fd 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.

Returns:

A pointer to the pathname for fd, or NULL if an error occurred (errno is set).

Errors:

EBADF The fd argument is invalid.
ENOSYS The ttyname() function isn’t implemented for the filesystem specified by fd.
ENOTTY Not a tty.
Examples:

```c
/*
 * 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

<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(), setsid(), ttyname_r()
ttname_r()

Get a fully qualified pathname for a file

Synopsis:

```c
#include <unistd.h>

int ttname_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 `ttname_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 `ttyname_r()` function isn’t implemented for the filesystem specified by `filedes`.
- **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

**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:**

ctermid(), errno, setsid(), ttyname()
**Synopsis:**
```
#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
tzset()

Set the time according to the current time zone

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(" day: %d\n", daylight );
    printf(" timezone: %ld\n", timezone );
    printf(" time zone names: %s %s",
           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

<table>
<thead>
<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:

cftime(), daylight, localtime(), localtime_r(), mktime(), strftime()
timezone, tzname

“Setting the time zone” in the Configuring Your Environment chapter of the Neutrino User’s Guide
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.

Errors:
EAGAIN All timers are in use; wait for a process to release one
and try again.

Examples:
#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>
</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:

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()
UDP
Internet User Datagram Protocol

Synopsis:

```c
#include <sys/socket.h>
#include <netinet/in.h>

int socket( AF_INET,
            SOCK_DGRAM,
            0 );
```

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.

ENOBUFFS  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.

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

Safety

Cancellation point  No

continued...
nulltoa(), ulltoa()

<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:

atoi(), atol(), itoa(), ltoa(), sscanf(), strtol(), strtoul(), utoa()
umask() Set the file-mode creation mask for the process

Synopsis:

```c
#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_IROGRP</td>
<td>S_IROTH</td>
<td>Read</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>

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_IXUSR | /* owner executable */
         | S_IRGRP | S_IWGRP /* group read write */
         | S_IXGRP; /* group executable */
    omask = umask( nmask );
    printf( "Mask changed from %o to %o\n",
            omask, nmask );
    return EXIT_SUCCESS;
}
```

**Classification:**

POSIX 1003.1

---

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Yes
- Thread: Yes
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>
<th></th>
</tr>
</thead>
</table>
| Cancellation point | Yes

continued...
umount()

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:

mount()

Writing a Resource Manager in *Programmer’s Guide*
UNALIGNED_PUT16()
Write a misaligned 16-bit value safely

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:

- `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>
<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_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()`
UNALIGNED_PUT32()
Write a misaligned 32-bit value safely

Synopsis:

```c
#include <gulliver.h>

void UNALIGNED_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 libc` option to `qcc` to link against this library. This library is usually included automatically.

Description:

The `UNALIGNED.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>
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_PUT64()`,
`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

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_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

<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_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 UNALIGNMENT 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 UNALIGNMENT_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_RET64()
UNALIGNED RET64()
Access a misaligned 64-bit value safely

Synopsis:
#include <gulliver.h>

uint64_t UNALIGNd_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

<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…
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()`, `ntohl()`,
`ntohs()`, `UNALIGNED_PUT16()`, `UNALIGNED_PUT32()`,
`UNALIGNED_PUT64()`, `UNALIGNED_RET16()`,
`UNALIGNED_RET32()`
uname()

Get information about the operating system

Synopsis:

```
#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:

```c
/*
 * 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

<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`
`uname` in the Utilities Reference
ungetc()

Push a character back onto an input stream

Synopsis:

```
#include <stdio.h>

int ungetc( int c,
            FILE *fp );
```

Arguments:

- `c` The character that you want to push back.
- `fp` 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 )
{
    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

<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()`, `getc()`, `getc_unlocked()`, `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 (`fseek()`, `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 `msg hdr` 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 **cmshdr** fields have
the following values:

```c
cmsg_len = sizeof(struct cmshdr) + 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.
### 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()`
**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

**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 `unsetenv()` function manipulates the environment pointed to by the global `environ` variable.

See also:

`clearenv()`, `getenv()`, `putenv()`, `setenv()`
Suspend a thread for a given number of microseconds

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.

`EINVAL` The `useconds` argument is too large.
Examples:

```c
#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[] )
```

November 2, 2006

C Library — T to Z 3639


```c
{ 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

<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, 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.
utimes()

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.

EROFS 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
• USERPROCESS
• DEAD_PROCESS
• ACCOUNTING

ut_exit
The exit status of a process marked as DEAD_PROCESS. The structure exit_status includes at least the following members:

• e_termination — the termination status.
• e_exit — the exit status.

ut_time
The time that this entry was made.

Classification:
Unix

See also:
endutent(), getutent(), getutid(), getutline(), pututline(), setutent(), utmpname()

login in the 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 `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 `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`

  Specifies the user information file.

**Classification:**

Unix
utmpname()

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:

endutent(), getutent(), getutid(), getutline(), pututline(), setutent(), utmp

login in the Utilities Reference
Convert an unsigned integer into a string, using a given base

Synopsis:

```c
#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:
  
  \[ 8 \times \text{sizeof}(\text{int}) + 1 \]

  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
", 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>
<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{atoi()}, \textit{atol()}, \textit{itoa()}, \textit{ltoa()}, \textit{sscanf()}, \textit{strtol()}, \textit{stroul()}, \textit{ultoa()}
Synopsis:

```
#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:

- `verr()` - `vscanf()` - `vsyslog()`
- `verrx()` - `vsllog()` - `vwarn()`
- `vfprintf()` - `vsnprintf()` - `vwarnx()`
- `vfscanf()` - `vsscanf()`
- `vfwprintf()` - `vsscanf()`
- `vfscanf()` - `vscanf()`
- `vfprintf()` - `vprintf()`
- `vfwscanf()` - `vfscanf()`
- `vfprintf()` - `vfscanf()`

**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));
}
```

# va_arg()

```c
void vfoo(char *fmt, va_list va) {
    handle_foo(fmt, &va);
}

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, ...) {
    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:**

```c
#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;
```
```c
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

continued...
va_arg() is a macro.

See also:
va_copy(), va_end(), va_start()
va_copy()

Make a copy of a variable argument list

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()**

**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_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

<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:

`va_end()` is a macro.

See also:

`va_arg()`, `va_copy()`, `va_start()`
Synopsis:

```c
#include <stdarg.h>

void va_start( va_list param,
               previous );
```

Arguments:

- `param` A `va_list` object that the "vaargs" 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()**

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>

Caveats:

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>
</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:

`memalign()`, `sysconf()`
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:

```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 `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
verr(), verrx()

- the string associated with the current value of errno
- 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

**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:**

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:

```
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:

- `EAGAIN`: The system-imposed limit on the total number of processes under execution would be exceeded. This limit is determined when the system is generated.
- `ENOMEM`: There isn’t enough memory for the new process.

Classification:

POSIX 1003.1 XSI

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>No</td>
</tr>
</tbody>
</table>

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 ioctls are allowed and input attempts result in an EOF indication.

See also:

`execve()`, `exit()`, `fork()`, `ioctl()`, `sigaction()`, `wait()`
vfprintf()

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

<table>
<thead>
<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:

errmsg, fprintf(), fwprintf(), printf(), snprintf(), sprintf(), swprintf(),
va_start(), vfwprintf(), 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, \textit{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( "\n%s, %s %d, %d\n",
        weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

\begin{tabular}{ll}
\hline
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & No \\
Thread & Yes \\
\hline
\end{tabular}
See also:

`errno`, `fscanf()`, `fwscanf()`, `scanf()`, `sscanf()`, `swscanf()`, `va_start()`,
`vfwscanf()`, `vscanf()`, `vsscanf()`, `vswscanf()`, `vwscanf()`, `wscanf()`
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

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, fprintf(), fwprintf(), printf(), snprintf(), sprintf(), swprintf(), va_start(), vfprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
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()

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, fscanf(), fwscanf(), scanf(), sscanf(), swscanf(), va_start(),
vfscanf(), vscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()
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**

<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`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`, `va_start()`, `vfprintf()`, `vfprintf()`, `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, *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];

    ffind( "%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

<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(), fswscanf(), scanf(), sscanf(), swscanf(), va_start(),
vfscanf(), vfwscanf(), vsscanf(), vswscanf(), vwscanf(), wscanf()
vslogf()

Send a message to the system logger (varargs)

Synopsis:
#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>
</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(), slogi(), slogf()`  
`slogger, sloginfo` in the Utilities Reference
vsnprintf()  
© 2006, QNX Software Systems GmbH & Co. KG.

Write formatted output to a character array, up to a maximum number of characters (varargs)

Synopsis:
```
#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 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()`.
- **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()`.
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;
}
```
vsnprintf()

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 `vsnprintf()` in a signal handler if the data isn’t floating point.

See also:

`errno`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`, `va_start()`, `vfprintf()`, `vfprintf()`, `vprintf()`, `vsprintf()`, `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

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Read the Caveats
- Thread: Yes
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()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, `vswprintf()`, `vwprintf()`, `wprintf()`
vsscanf()  © 2006, QNX Software Systems GmbH & Co. KG.
Scan input from a string (varargs)

Synopsis:
```
#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( "\n%s, %s %d, %d\n",
            weekday, month, day, year );
    return EXIT_SUCCESS;
}
```

Classification:

ANSI, POSIX 1003.1

Safety

- Cancellation point: No
- Interrupt handler: No
- Signal handler: Read the Caveats
- Thread: Yes
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()`, `vswscanf()`, `vwscanf()`, `wscanf()`
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()`.
vswprintf()

Returns:
The number of wide characters written, excluding the terminating NULL, 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>Read the Caveats</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(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf(), wprintf()
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.
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:

```c
#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()`.
**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.
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>
<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()`, `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).
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`, `fprintf()`, `fwprintf()`, `printf()`, `snprintf()`, `sprintf()`, `swprintf()`,
`va_start()`, `vfprintf()`, `vfwprintf()`, `vprintf()`, `vsnprintf()`, `vsprintf()`,
`vswprintf()`, `wprintf()`
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()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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`, `scanf`, `sscanf`, `swscanf`, `va_start`, `vfscanf`, `vfwscanf`, `vscanf`, `vsscanf`, `vswscanf`, `wscanf`
**wait()**

wait for the status of a terminated child process

**Synopsis:**

```c
#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:
\textbf{WEXITSTATUS( stat\_val )}

Evaluates to the low-order 8 bits of the termination status of the child process if the value of \textbf{WIFEXITED( stat\_val )} is nonzero.

\textbf{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.

\textbf{WIFEXITED( stat\_val )}

Evaluates to a nonzero value if the status returned was from a normally terminated child process.

\textbf{WIFSIGNALED( stat\_val )}

Evaluates to nonzero value if the child process terminated from reception of a signal that wasn’t caught.

\textbf{WIFSTOPPED( stat\_val )}

Evaluates to a nonzero value if the status returned is for a child process that’s stopped.

\textbf{WSTOPSIG( stat\_val )}

Evaluates to the number of the signal that caused the child process to stop if the value of \textbf{WIFSTOPPED( stat\_val )} is nonzero.

\textbf{WTERMSIG( stat\_val )}

Evaluates to the number of the signal that terminated the child process if the value of \textbf{WIFSIGNALED( stat\_val )} is nonzero.

This macro isn’t part of a POSIX standard:

\textbf{WCOREDUMP( stat\_val )}

Evaluates to a nonzero value if the child process left a core dump.

One of the macros \textbf{WIFEXITED(*stat\_loc)} and \textbf{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

<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`, `spawn()`, `wait4()`, `waitid()`, `waitpid()`
wait3()

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.
  - 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.

- **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.
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

<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:

New applications should use `waitpid()`.

See also:

`exit()`, `fork()`, `pause()`, `wait4()`, `waitid()`, `waitpid()`
**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:

\[
\text{wait3( stat\_loc, options, resource\_usage );}
\]

is equivalent to the call:

\[
\text{waitpid( (pid\_t)-1, stat\_loc, options );}
\]
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:

```c
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.

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:**

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...
wait4()

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:

exit(), fork(), pause(), wait(), wait3(), waitid(), waitpid()
**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.
waitid()

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:

execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(),
execvpe(), exit(), fork(), pause(), sigaction(), signal(), wait(), wait3(),
wait4(), waitpid()
waitpid()  

Suspends 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()**

**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 *statloc* is undefined.

EINVAL    The value of the *options* argument isn’t valid.

**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:**

*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:

- 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(), verr(), verxx(), 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.
**wcrtomb()**

© 2006, QNX Software Systems GmbH & Co. KG.

**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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>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.
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...
See also:

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**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

**Safety**

- Cancellation point: No
- Interrupt handler: Yes

*continued…*
**wcschr()**

© 2006, QNX Software Systems GmbH & Co. KG.

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

**See also:**

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `wcscspn()`, `wcspbrk()`, `wcsrchr()`, `wcsspn()`, `wcsstr()`, `wcstok()`
Synopsis:

```
#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-characters 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

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

continued…
### wcsncmp()

© 2006, QNX Software Systems GmbH & Co. KG.

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

See also:

- `strcasecmp()`, `strcmp()`, `strcoll()`, `stricmp()`, `strncasecmp()`, `strncmp()`, `strnicmp()`, `wcscoll()`, `wcsncmp()`
Synopsis:

#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
wcscoll()

See also:

setlocale(), strcasecmp(), strcmp(), strcoll(), stricmp(), strncasecmp(), strncmp(), strnicmp(), wcscmp(), wcscmp()
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
wcsncpy()

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:

memmove(), strcpy(), strdup(), strncpy(), wcsncpy(), wmemmove()
wcscspn()
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:

```c
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

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
</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:

`memchr()`, `strchr()`, `strcspn()`, `strchr()`, `strspn()`, `strstr()`,
`strtok()`, `strtok_r()`, `wcschr()`, `wcscrear()`, `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.
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`, `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

<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:

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>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:

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:

```c
#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 `-l c` 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()

© 2006, QNX Software Systems GmbH & Co. KG.

#### 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:

strcasecmp(), strcmp(), strcmpeq(), strcoll(), stricmp(), strncasecmp(), strncmp(), strncmpi(), wcscmp(), wcscoll()
**wcsncpy()**

Copy a wide-character string, to a maximum length

**Synopsis:**

```c
#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.
wcsncpy()

Returns:

The same pointer as \textit{\textit{ws1}}.

Classification:

ANSI, POSIX 1003.1

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{Safety} & \\
\hline
Cancellation point & No \\
Interrupt handler & Yes \\
Signal handler & Yes \\
Thread & Yes \\
\hline
\end{tabular}
\end{center}

See also:

\textit{memmove()}, \textit{strcpy()}, \textit{strdup()}, \textit{strncpy()}, \textit{wcscpy()}, \textit{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

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()`, `strrchr()`, `strspn()`, `strstr()`,
`strtok()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcsrchr()`, `wcsspn()`, `wcsstr()`,
`wcstok()`
**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

**Safety**

- Cancellation point: No
- Interrupt handler: Yes

...continued
### wcsrchr()

#### 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()`, `strchr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcspbrk()`, `wcspn()`, `wcsstr()`, `wcstok()`


**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:**

```
libc
```

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.

- 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 `(size_t)-1` if an invalid wide-character code was found.

**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.


Synopsis:

```c
#include <wchar.h>

size_t wcsspn( 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 `wcsspn()` 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

<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:

`memchr()`, `strchr()`, `strcsnp()`, `strpbrk()`, `strrchr()`, `strspn()`, `strsr()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcscspn()`, `wcspbrk()`, `wcsrchr()`, `wcsstr()`, `wcstok()`
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
Source: C Library — T to Z

wsstr()

Safety

| Signal handler | Yes |
| Thread         | Yes |

See also:

memchr(), strchr(), strcspn(), strpbrk(), strrev(), strspn(), strstr(), strtok(), strtok_r(), wcschr(), wcscspn(), wcspbrk(), wcsrchr(), wcscspn(), wcstok()
Convert a wide-character string into a double, float, or long double

Synopsis:

```c
#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

<table>
<thead>
<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_

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
wcstomax(), wcstoulmax()

Convert a wide-character string into an integer

Synopsis:

```c
#include <inttypes.h>

intmax_t wcstomax ( 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.
wcstomax(), wcstoumax()

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{wcstomax()} and \texttt{wcstoumax()} functions are the same as the \texttt{wcstol()}, \texttt{wcstoll()}, \texttt{wcstoul()}, and \texttt{wcstoull()} functions except that they return objects of type \texttt{intmax_t} and \texttt{uintmax_t}.

Returns:

The converted value.

If the correct value causes an overflow, (\texttt{INTMAX_MAX} | \texttt{UINTMAX_MAX} or \texttt{INTMAX_MIN}) is returned according to the sign 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{EINVAL}.

Classification:

ANSI, POSIX 1003.1

\begin{center}
\begin{tabular}{lc}
\hline
\textbf{Safety} & \\
Cancellation point & No \\
Interrupt handler & Yes \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}
\end{center}

See also:

\texttt{strtol()}, \texttt{wcstol()}, \texttt{wcstoul()}

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
**wcstok()**

Break a wide-character string into tokens

**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:**

`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 *ws2 may differ from call to call. When no tokens
remain in *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

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:

`memchr()`, `strchr()`, `strcspn()`, `strpbrk()`, `strrchr()`, `strset()`, `strspn()`,
`strstr()`, `strtok()`, `strtok_r()`, `wcschr()`, `wcsccspn()`, `wcschr()`, `wcschr()`,
`wcsspn()`, `wcsstr()`
Convert a wide-character string into a long integer

Synopsis:

```c
#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 `wcstol()` function converts the string pointed to by `ptr` into a `long`; `wcstoll()` converts the string into a `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 `endptr` isn’t NULL, a pointer to the unrecognized wide 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`.

Errors:

```
ERANGE The value is not representable
EINVVAL The value for `base` is not supported or no conversion could be performed.
```
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:

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()
**wcstoul(), wcstoull()**

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:

- `wcstoul()` converts the string into an `unsigned long`
- `wcstoull()` converts it into a `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 `endptr` points to, if `endptr` isn’t NULL.

If `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 `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 `base` are permitted. If the value of `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, `ULONG_MAX` or `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`. 
Errors:

ERANGE The value is not representable
EINVVAL The value for base is not supported or no conversion could be performed.

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:

errno

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
Synopsis:
```
#include <wchar.h>

int wcscxfrm( 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 `wcscxfrm()` 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 `wcscxfrm()` 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 `wcscxfrm()`, 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.

Classification:

ANSI, POSIX 1003.1

\textit{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:

\textit{setlocale()}, \textit{strxfrm}()
**Synopsis:**

```c
#include <wchar.h>

int wctob( wchar_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>

November 2, 2006
See also:

“String manipulation functions” and “Wide-character functions” in the summary of functions chapter.
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 mbbuffer[MB_CUR_MAX];

int main( void )
{
    int len;

    printf( "Character encodings do %shave "
            "state-dependent \nencoding.\n",
            ( wctomb( NULL, 0 ) )  
            ? "" : "not " );

    len = wctomb( mbbuffer, wchar );
    mbbuffer[len] = '\0';
    printf( "%s(%d)\n", mbbuffer, len );
    return EXIT_SUCCESS;
}
```

This produces the output:

Character encodings do not have state-dependent encoding.

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:

“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

<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:

towctrans()

String manipulation functions
Wide-character functions
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`
- `digit`
- `punct`
- `alpha`
- `graph`
- `space`
- `blank`
- `lower`
- `upper`
- `cntrl`
- `print`
- `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()

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 `-l c` 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:

memccpy(), memcmp(), memcpy(),memcmp(), memmove(), memset()
wcschr(), wcsrchr(), 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
### wmempcmp()

**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()`, `memcmp()`, `memcpy()`, `memicmp()`, `memmove()`, `memset()`

`wcsncmp()`, `wcscmp()`, `wmemchr()`, `wmempcpy()`, `wmempmove()`, `wmempset()`
Copy wide characters from one buffer to another

Synopsis:

```c
#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 `-l c` 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.
wmemcpy()  

Returns:  
A pointer to the destination buffer (i.e the same pointer as \textit{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>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:  

\textit{memccpy()}, \textit{memcmp()}, \textit{memcpy()}, \textit{memicmp()}, \textit{memmove()}, \textit{memset()}
\textit{wcscpy()}, \textit{wcsncmp()}, \textit{wmemchr()}, \textit{wmemcmpl()}, \textit{wmemmove()}, \textit{wmemset()}

November 2, 2006
**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 `wmemcpyp()` for greater speed when copying buffers that don’t overlap.

### Returns:

A pointer to the destination buffer (i.e. the same pointed 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()
wmemchr(), wmemcmp(), wmemcpy(), wmemset()
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:

```c
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>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:

`memccpy()`, `memcpy()`, `memcmp()`, `memmove()`, `memset()`,
`wmemchr()`, `wmemcpy()`, `wmemcmp()`, `wmemmove()`
**Synopsis:**

```c
#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.

 dép 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()

<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:

glob(), globfree(), wordfree()
Free a word expansion buffer

Synopsis:

```
#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
```

<table>
<thead>
<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()`
**wprintf()**

Write formatted output to stdout

**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 `-l c` 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

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
</table>

continued...
wprintf()  

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, fprintf(), fwprintf(), printf(), snprintf(), sprintf(), swprintf(), vfprintf(), vfwprintf(), vprintf(), vsnprintf(), vsprintf(), vswprintf(), vwprintf()
write()  © 2006, QNX Software Systems GmbH & Co. KG.

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.
write()

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 than `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`. 

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_ctime` 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 filedes.

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

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(), writev()
writeblock()
Write blocks of data to a file

Synopsis:

#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.
umblks   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.

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

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:

open(), readblock(), write()
**writev()**

Write bytes to a file

**Synopsis:**

```c
#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.
writev()

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 \texttt{write()} function isn’t implemented for the filesystem specified by \texttt{fdedes}.

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.

\textbf{Classification:}

POSIX 1003.1 XSI

\begin{tabular}{ll}
\textbf{Safety} & \\
Cancellation point & Yes \\
Interrupt handler & No \\
Signal handler & Yes \\
Thread & Yes \\
\end{tabular}

\textbf{See also:}

\texttt{close()}, \texttt{creat()}, \texttt{dup()}, \texttt{dup2()}, \texttt{errno}, \texttt{fcntl()}, \texttt{lseek()}, \texttt{open()}, \texttt{pipe()}, \texttt{read()}, \texttt{readv()}, \texttt{select()}, \texttt{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


**wscanf()**

© 2006, QNX Software Systems GmbH & Co. KG.

<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(), sscanf(), swscanf(), vfscanf(), vfwscanf(), vscanf(), vsscanf(), vswscanf(), vwscanf()
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:

```c
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**

<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(), j1(), jn(), y1(), yn()
**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 );
    
    // Compute y1 function
    y = y1( x );
    
    // Compute y1f function
    z = y1f( x );
}
```
```c
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;
```

Classification:

`y1()` is POSIX 1003.1 XSI; `y1f()` 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:

`errno, j0(), j1(), jn(), y0(), yn()`
Compute a Bessel function of the second kind

Synopsis:
```
#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()

<table>
<thead>
<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, j0(), j1(), jn(), y0(), y1()
Appendix A

SOCKS — A Basic Firewall

In this appendix...

About SOCKS 3823
How to SOCKSify a client 3823
What SOCKS expects 3824
About SOCKS

SOCKS is a package consisting of a proxy server, client programs (rftpd 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 “SOCKS_client” default string).

2. Add the following options to your compile commands:

   -Dconnect=Rconnect -Dgetsockname=Rgetsockname
   -Dbind=Rbind -Daccept=Raccept -Dlisten=Rlisten
   -Drmd=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

The SOCKS library covers only some of the socket functions, which must be called in a particular order:

You must use TCP; SOCKS doesn’t support UDP.

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

Third-Party Copyright Notices
BSD stack

Copyright © 1997 Christopher G. Demetriou.
All rights reserved.
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
MERCHANDIBILITY 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.
Appendix C

Summary of Safety Information

In this appendix...

Cancellation points  3845
Interrupt handlers  3850
Signal handlers  3853
Multithreaded programs  3867
Cancellation points

The following functions are cancellation points:

ConnectAttach()    close()
ConnectAttach_r()  closedir()
ConnectDetach()    closelog()
ConnectDetach_r()  connect()
ConnectServerInfo()  creat()
ConnectServerInfo_r()  creat64()
InterruptWait()    delay()
InterruptWait_r()  devctl()
MsgSend()    dispatch_block()
MsgSend_r()    dispatch_unblock()
MsgSendsv()    dclclose()
MsgSendsv_r()  dlopen()
MsgSendv()    ds_clear()
MsgSendv_r()    ds_create()
MsgSendvs()    ds_deregister()
MsgSendvs_r()  ds_flags()
SignalSuspend()    ds_get()
SignalSuspend_r()  ds_register()
SignalWaitinfo()    ds_set()
SignalWaitinfo_r()  endgrent()
SyncCondvarSignal()    endhostent()
SyncCondvarSignal_r()  endnetent()
SyncCondvarWait()    endprotoent()
SyncCondvarWait_r()  endservent()
ThreadJoin()    endsvent()
ThreadJoin_r()  endspent()
accept()    endutent()
aio_suspend()    eof()
cache_fini()    err()
cache_init()    errx()
cfgopen()    fcfgopen()
chsize()    fchown()
clock_nanosleep()    fclose()
Cancellation points

fcloseall()
fdoopen()
fflush()
fgetc()
fgetchar()
fgets()
fgetspentr()
fgetwc()
fgetsws()
flushall()
freopen()
freopen64()
forkpty()
fprintf()
fputc()
fputchar()
fputs()
fputwc()
fputws()
freopen()
freopen64()
getaddrinfo()
getc()
getchar()
getcwd()
getgrent()
getgrgid()
getgrgid_r()
getgrnam()
getgrnam_r()
getgrouplist()
gethostbyaddr()
gethostbyaddr_r()
gethostbyname()
gethostbyname2()
gethostbyname_r()
gethostent()
gethostent_r()
gethostid() gethostname()
getifaddrs()
getlogin()
getlogin_r()
getlogin_r()
getlogin()
getnameinfo()
getnetbyaddr()
getnetbyname()
getnetbyname2()
getnetbyname_r()
getnetent()
getopt()
getpass()
getpeername()
getprotoent()
getprotoent_r()
getprotoent() getprotoent()
getprotobyname()
getprotobynumber()
getprotobynumber() getpropbyname() getprotobyname() getprotobyname()
getpropbyname() getprotobyname()
getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
getpropbyname() getpropbyname() getpropbyname()
Cancellation points

getsockopt()  
getsockopt()  
getspent()  
getspent_r()  
getspnam()  
getsparnam_r()  
getutent()  
getutid()  
getutline()  
getw()  
getwc()  
getwchar()  
getwd()  
glob()  
herror()  
if__indextoname()  
if__nameindex()  
if__nameindex()  
initstate()  
input_line()  
ioctl_attr_lock()  
ioctl_attr_frylock()  
isfdtype()  
listen()  
login_tty()  
ltruncate()  
message_attach()  
message_connect()  
message_detach()  
mknod()  
mkstemp()  
mktemp()  
modem_open()  
modem_read()  
modem_script()  
modem_write()  
mount()  
msync()  
mq_receive()  
mq_send()  
mq_timedreceive()  
mq_timedsend()  
mq_timedreceive()  
mq_timedsend()  
name_attach()  
name_close()  
name_detach()  
name_open()  
nanosleep()  
nap()  
napms()  
nbacnnnect()  
nbacnnnect_result()  
netmgr_ndtostr()  
netmgr_strtond()  
nftw()  
nftw64()  
open()  
open64()  
opendir()  
openfd()  
openlog()  
opempt()  
pathfind()  
pathfind_r()  
pathmgr_symlink()  
pathmgr_unlink()  
pause()  
ppccard_arm()  
ppccard_attr()  
ppccard_detach()  
ppccard_attr()  
ppccard_detach()  
ppccard_info()  
ppccard_lock()  
ppccard_raw_read()  
ppccard_unlock()  
pci_attr()
Cancellation points

pci_attach_device()  pulse_attach()
pci_detach()          putchar()
pci_detach_device()   putc_unlocked()
pci_find_class()      putchar()
pci_find_device()     putchar_unlocked()
pci_irq_routing_options() putspendt()
pci_map_irq()         pututline()
present()             putw()
pci_read_config()     pwrite()
pci_read_config16()   putwc()
pci_read_config32()   putwchar()
pci_read_config8()    pwrite64()
pci_rescan_bus()      pwrite64()
pci_write_config()    read()
pci_write_config16()  read()
pci_write_config32()  read_main_config_file()
pci_write_config8()   readblock()
pclose()              readcond()
error()               readdir()
poll()                readv()
popen()               realpath()
pread()               recv()
pread64()             recvfrom()
printf()              recvmsg()
pthread_cond_timedwait() remove()
pthread_cond_wait()   rename()
pthread_join()        res_init()
pthread_rwlock_rdlock() res_mkquery()
pthread_rwlock timedrdlock() res_query()
pthread_rwlock timedwrlock() res_querydomain()
pthread_rwlock tryrdlock() res_search()
pthread_rwlock trywrlock() res_send()
pthread_rwlock wrlock() resmgr_attach()
pthread_sleepon_lock() resmgr_block()
pthread_sleepon timedwait() resmgr_context_alloc()
pthread_sleepon_wait() resmgr_context_free()
pthread_testcancel()  resmgr_detach()
Cancellation points

resmgr_devino() setsockopt()
resmgr_handler() setutent()
rewind()shm_ctl()
rewinddir()shm_ctl_special()
rrresvport()shutdown()
rsrcregmgr_attach() sigpause()
rsrcregmgr_create() sigsuspend()
rsrcregmgr_destroy() sigtimedwait()
rsrcregmgr_detach() sigwait()
rsrcregmgr_devno_attach() sigwaitinfo()
rsrcregmgr_devno_detach() sleep()
rsrcregmgr_query() slogb()
ruserok() slogf()
scandir() slogi()
scanf() snmp_close()
scip_bindx() snmp_open()
scip_connectx() snmp_read()
scip_getladdr() snmp_send()
scip_getpaddr() snmp_timeout()
scip_peeloff() socketatmark()
scip_recvmsg() socket()
scip_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()
setprotoent() thread_pool_start()
setpwent() tmpfile()
setservent() tmpfile64()
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()
atol()  isalnum()
atoll()  isalpha()
atomic_add()  isascii()
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()  outbe16()
iswcntrl()  outle16()
iswdigit()  outle32()
iswgraph()  rindex()
iswlower()  setdomainname()
iswprint()  sigaddset()
iswpunct()  sigdelset()
iswspace()  sigemptyset()
iswupper()  sigfillset()
iswdigit()  sigismember()
isxdigit()  sigmask()
itoa()  straddstr()
lltoa()  strcasecmp()
lsearch()  strcat()
ltoa()  strchr()
max()  strcmp()
memccpy()  strcmp()
memchr()  strcoll()
memcmp()  strcpy()
memcpy()  strcsppn()
memcpyv()  stricmp()
memiccmp()  strlen()
memmove()  strlwr()
memset()  strncasecmp()
mini()  strncat()
nanospin_count()  strncmp()
nsec2timespec()  strncpy()
ntohl()  strnicmp()
ntohs()  strnset()
offsetof()  strpbrk()
out16()  strrrchr()
out16s()  strrev()
out32()  strsep()
out32s()  strset()
out8()  strspn()
out8s()  strstr()
outbe16()  strtoimax()
<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>strtok_r()</td>
<td>wcscmp()</td>
</tr>
<tr>
<td>strtol()</td>
<td>wcscpy()</td>
</tr>
<tr>
<td>strtoll()</td>
<td>wcscspn()</td>
</tr>
<tr>
<td>strtoul()</td>
<td>wcslen()</td>
</tr>
<tr>
<td>strtoull()</td>
<td>wcscat()</td>
</tr>
<tr>
<td>strtoumax()</td>
<td>wcscmp()</td>
</tr>
<tr>
<td>strupr()</td>
<td>wcscpy()</td>
</tr>
<tr>
<td>strxfrm()</td>
<td>wcspbrk()</td>
</tr>
<tr>
<td>swab()</td>
<td>wcsrchr()</td>
</tr>
<tr>
<td>timespec2nsec()</td>
<td>wcsspnp()</td>
</tr>
<tr>
<td>tolower()</td>
<td>wcsstr()</td>
</tr>
<tr>
<td>toupper()</td>
<td>wcstomax()</td>
</tr>
<tr>
<td>towctrans()</td>
<td>wcstol()</td>
</tr>
<tr>
<td>towlower()</td>
<td>wcstoll()</td>
</tr>
<tr>
<td>towupper()</td>
<td>wcstoul()</td>
</tr>
<tr>
<td>ultoa()</td>
<td>wcstoul()</td>
</tr>
<tr>
<td>utoa()</td>
<td>wcstoumax()</td>
</tr>
<tr>
<td>va_arg()</td>
<td>wctype()</td>
</tr>
<tr>
<td>va_copy()</td>
<td>wmemchr()</td>
</tr>
<tr>
<td>va_end()</td>
<td>wmemcmp()</td>
</tr>
<tr>
<td>va_start()</td>
<td>wmemcpy()</td>
</tr>
<tr>
<td>wcscat()</td>
<td>wmemmove()</td>
</tr>
<tr>
<td>wcschr()</td>
<td>wmemset()</td>
</tr>
</tbody>
</table>

See the “Caveats” section for the following functions for more information:

- TraceEvent()
- nanospin()
- nanospin_ns()
- nanospin_ns_to_count()

**Signal handlers**

It’s safe to call the following functions from a signal handler:
CACHETYPE_FLUSH()       ENDIAN_SWAP16()
CACHETYPE_INVAL()       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()
DebugKDOoutput()        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 v_r() SchedGet r()
MsgReceive r() SchedInfo(
MsgReceivev() SchedInfo r()
MsgReceivev_r() SchedSet(
MsgReply() SchedSet r()
MsgReply r() SchedYield(
MsgReplyv() SchedYield r()
MsgReplyv_r() SignalAction(
MsgSend() SignalAction r()
MsgSendPulse() SignalKill(
MsgSendPulse r() SignalKill r()
MsgSend r() SignalProcmask(
MsgSendnc() SignalProcmask r()
MsgSendnc_r() SignalSuspend(
MsgSendnv() SignalSuspend r()
MsgSendnv_r() SignalWaitinfo(
MsgSendsv() SignalWaitinfo r()
MsgSendsvnc() SyncCondvarSignal(
MsgSendsvnc_r() SyncCondvarSignal r()
MsgSendv() SyncCondvarWait(
MsgSendv r() SyncCondvarWait r()
MsgSendvnc() SyncCondvarWait r()
MsgSendvnc r() SyncCtl(
MsgSendvs() SyncCtl r()
MsgSendvs_r() SyncDestroy(
MsgSendvsnc() SyncDestroy r()
MsgSendvsnc_r() SyncMutexEvent(
MsgVerifyEvent() SyncMutexEvent r()
MsgVerifyEvent r() SyncMutexLock(
MsgWrite() SyncMutexLock r()
MsgWrite r() SyncMutexRevive(
MsgWritev() SyncMutexRevive r()
MsgWritev_r() SyncMutexUnlock(
ND_NODE_CMP() SyncMutexUnlock r()
SETOV() SyncSemPost(
SYSPAGE_CPU_ENTRY() SyncSemPost r()
SYSPAGE_ENTRY() SyncSemWait(
SchedGet() SyncSemWait r())
Signal handlers

\texttt{SyncTypeCreate()} \quad \texttt{\_intr\_y86()}
\texttt{SyncTypeCreate}_r() \quad \texttt{\_sfree()}
\texttt{ThreadCancel()} \quad \texttt{abs()}
\texttt{ThreadCancel}_r() \quad \texttt{access()}
\texttt{ThreadCreate()} \quad \texttt{aio\_cancel()}
\texttt{ThreadCreate}_r() \quad \texttt{aio\_error()}
\texttt{ThreadCtl()} \quad \texttt{aio\_fsync()}
\texttt{ThreadCtl}_r() \quad \texttt{aio\_read()}
\texttt{ThreadDestroy()} \quad \texttt{aio\_return()}
\texttt{ThreadDestroy}_r() \quad \texttt{aio\_suspend()}
\texttt{ThreadDetach()} \quad \texttt{aio\_write()}
\texttt{ThreadDetach}_r() \quad \texttt{alarm()}
\texttt{ThreadJoin()} \quad \texttt{alloca()}
\texttt{ThreadJoin}_r() \quad \texttt{alphasort()}
\texttt{TimerAlarm()} \quad \texttt{asctime()}
\texttt{TimerAlarm}_r() \quad \texttt{asctime\_r()}
\texttt{TimerCreate()} \quad \texttt{atoh()}
\texttt{TimerCreate}_r() \quad \texttt{atoi()}
\texttt{TimerDestroy()} \quad \texttt{atol()}
\texttt{TimerDestroy}_r() \quad \texttt{atoll()}
\texttt{TimerInfo()} \quad \texttt{atomic\_add()}
\texttt{TimerInfo}_r() \quad \texttt{atomic\_add\_value()}
\texttt{TimerSettime()} \quad \texttt{atomic\_clr()}
\texttt{TimerSettime}_r() \quad \texttt{atomic\_clr\_value()}
\texttt{TimerTimeout()} \quad \texttt{atomic\_set()}
\texttt{TimerTimeout}_r() \quad \texttt{atomic\_set\_value()}
\texttt{TraceEvent()} \quad \texttt{atomic\_sub()}
\texttt{\_UNALIGNED\_PUT16()} \quad \texttt{atomic\_sub\_value()}
\texttt{\_UNALIGNED\_PUT32()} \quad \texttt{atomic\_toggle()}
\texttt{\_UNALIGNED\_PUT64()} \quad \texttt{atomic\_toggle\_value()}
\texttt{\_UNALIGNED\_RET16()} \quad \texttt{basename()}
\texttt{\_UNALIGNED\_RET32()} \quad \texttt{bcmp()}
\texttt{\_UNALIGNED\_RET64()} \quad \texttt{bcopy()}
\texttt{\_RESMGR\_NPARTS()} \quad \texttt{bsearch()}
\texttt{\_RESMGR\_PTR()} \quad \texttt{btowc()}
\texttt{\_RESMGR\_STATUS()} \quad \texttt{bzero()}
\texttt{\_exit()} \quad \texttt{cache\_fini()}

3856 Appendix: C • Summary of Safety Information November 2, 2006
<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cache_init()</code></td>
<td><code>ds_register()</code></td>
</tr>
<tr>
<td><code>cfgetispeed()</code></td>
<td><code>ds_set()</code></td>
</tr>
<tr>
<td><code>cfgetospeed()</code></td>
<td><code>dup()</code></td>
</tr>
<tr>
<td><code>cfopen()</code></td>
<td><code>dup2()</code></td>
</tr>
<tr>
<td><code>cfmakeraw()</code></td>
<td><code>eaccess()</code></td>
</tr>
<tr>
<td><code>cfsetispeed()</code></td>
<td><code>encrypt()</code></td>
</tr>
<tr>
<td><code>cfsetospeed()</code></td>
<td><code>eof()</code></td>
</tr>
<tr>
<td><code>chdir()</code></td>
<td><code>errno()</code></td>
</tr>
<tr>
<td><code>chmod()</code></td>
<td><code>errx()</code></td>
</tr>
<tr>
<td><code>chown()</code></td>
<td><code>execle()</code></td>
</tr>
<tr>
<td><code>chsize()</code></td>
<td><code>execv()</code></td>
</tr>
<tr>
<td><code>clock()</code></td>
<td><code>execvpe()</code></td>
</tr>
<tr>
<td><code>clock_getcpuclockid()</code></td>
<td><code>fcfgopen()</code></td>
</tr>
<tr>
<td><code>clock_getres()</code></td>
<td><code>fchmod()</code></td>
</tr>
<tr>
<td><code>clock_gettime()</code></td>
<td><code>fchown()</code></td>
</tr>
<tr>
<td><code>clock_nanosleep()</code></td>
<td><code>fcntl()</code></td>
</tr>
<tr>
<td><code>clock_settime()</code></td>
<td><code>fdatasync()</code></td>
</tr>
<tr>
<td><code>close()</code></td>
<td><code>ffs()</code></td>
</tr>
<tr>
<td><code>confstr()</code></td>
<td><code>fileno()</code></td>
</tr>
<tr>
<td><code>creat()</code></td>
<td><code>flink()</code></td>
</tr>
<tr>
<td><code>creat64()</code></td>
<td><code>flock()</code></td>
</tr>
<tr>
<td><code>ctime()</code></td>
<td><code>fnmatch()</code></td>
</tr>
<tr>
<td><code>ctime_r()</code></td>
<td><code>fork()</code></td>
</tr>
<tr>
<td><code>daemon()</code></td>
<td><code>forkpty()</code></td>
</tr>
<tr>
<td><code>delay()</code></td>
<td><code>fpathconf()</code></td>
</tr>
<tr>
<td><code>devctl()</code></td>
<td><code>fseek()</code></td>
</tr>
<tr>
<td><code>dirname()</code></td>
<td><code>fcntl()</code></td>
</tr>
<tr>
<td><code>dispatch_block()</code></td>
<td><code>ftime()</code></td>
</tr>
<tr>
<td><code>dispatch_unblock()</code></td>
<td><code>ftruncate()</code></td>
</tr>
<tr>
<td><code>div()</code></td>
<td><code>ftruncatemax()</code></td>
</tr>
<tr>
<td><code>dn_comp()</code></td>
<td><code>ftruncate64()</code></td>
</tr>
<tr>
<td><code>dn_expand()</code></td>
<td><code>ftruncate64()</code></td>
</tr>
<tr>
<td><code>ds_clear()</code></td>
<td><code>fsync()</code></td>
</tr>
<tr>
<td><code>ds_create()</code></td>
<td><code>ftime()</code></td>
</tr>
<tr>
<td><code>ds_deregister()</code></td>
<td><code>ftruncate()</code></td>
</tr>
<tr>
<td><code>ds_flags()</code></td>
<td><code>ftruncatemax64()</code></td>
</tr>
<tr>
<td><code>ds_get()</code></td>
<td><code>ftruncatemax64()</code></td>
</tr>
</tbody>
</table>
Signal handlers

iofunc_attr_ftrylock()  iofunc_openfd()
iofunc_attr_unlock()  iofunc_openfd_default()
iofunc_check_access()  iofunc_pathconf()
iofunc_chmod()  iofunc_pathconf_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_fdiinfo()  iofunc_time_update()
iofunc_fdiinfo_default()  iofunc_unblock()
iofunc_func_init()  iofunc_unblock_default()
iofunc_link()  iofunc_unlink()
iofunc_lock()  iofunc_unlock_ocb_default()
iofunc_lock_alloc()  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()  isfctype()
iofunc_ocb_attach()  isgraph()
iofunc_ocb_alloc()  islower()
iofunc_ocb_detach()  isprint()
iofunc_ocb_free()  ispunct()
iofunc_open()  isprint()
iofunc_open_default()  isupper()
Signal handlers

- iswalnum()
- iswalpha()
- iswcntrl()
- iswctype()
- iswdigit()
- iswgraph()
- iswlower()
- iswprint()
- iswpunct()
- iswspace()
- iswupper()
- iswxdigit()
- isxdigit()
- itoa()
- kill()
- killpg()
- labs()
- lchown()
- ldif()
- lfind()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
- login_tty()
- longjmp()
- lsearch()
- link()
- lio_listio()
- ltoa()
- localtime()
- lockf()
Signal handlers

name_open()
nanospin()
nanospin_calibrate()
nanospin_count()
nanospin_ns()
nanospin_ns_to_count()
nap()
napms()
nbaconnect_result()
netmgr_ntohlstr()
netmgr_remote_out()
netmgr_strtond()
nftw()
nftw64()
nice()
nsec2timespec()
ntohl()
ntohs()
offsetof()
offsetof()
op()}
open()
open64()
openfd()
openpty()
out16()
out16s()
out32()
out32s()
out8()
out8s()
outbe16()
outbe32()
outle16()
outle32()
pathconf()
pathfind()
pathmgr_symlink()
pathmgr.unlink()
pause()
pccard_arm()
pccard_attach()
pccard_detach()
pccard_info()
pccard_lock()
pccard_raw_read()
pccard_unlock()
pci_attach()
pci_attach_device()
pci_detach()
pci_detach_device()
pci_find_class()
pci_find_device()
pci_irq_routing_options()
pci_map_irq()
pci_present()
pci_read_config()
pci_read_config16()}
pci_read_config32()
pci_read_config8()
pci_rescan_bus()
pci_write_config()
pci_write_config16()}
pci_write_config32()
pci_write_config8()
pipe()
posix_mem_offset()
posix_mem_offset64()
posix_memalign()
pread()
pread64()
procmgr_daemon()
procmgr_event_notify()
procmgr_event_trigger()
procmgr_guardian()
Signal handlers

pthread_abort()
pthread_atfork()
pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_getguardsize()
pthread_attr_getinherithreds()
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_getshared()
pthread_barrierattr_init()
pthread_barrierattr_setshared()
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

```c
call

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_comp()
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() readdir()
pthread_setcancelstate() rename()
pthread_setcanceltype() resmgr_msgread()
pthread_setsconcurrenc() 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_destory()
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()
```
Signal handlers

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sched_getscheduler()</td>
<td>sigaddset()</td>
</tr>
<tr>
<td>sched_rr_get_interval()</td>
<td>sigblock()</td>
</tr>
<tr>
<td>sched_setparam()</td>
<td>sigdelset()</td>
</tr>
<tr>
<td>sched_setscheduler()</td>
<td>sigemptyset()</td>
</tr>
<tr>
<td>sched_yield()</td>
<td>sigfillset()</td>
</tr>
<tr>
<td>sem_close()</td>
<td>sigismember()</td>
</tr>
<tr>
<td>sem_destroy()</td>
<td>siglongjmp()</td>
</tr>
<tr>
<td>sem_getvalue()</td>
<td>sigmask()</td>
</tr>
<tr>
<td>sem_open()</td>
<td>signal()</td>
</tr>
<tr>
<td>sem_post()</td>
<td>sigpause()</td>
</tr>
<tr>
<td>sem_timedwait()</td>
<td>sigpending()</td>
</tr>
<tr>
<td>sem_trywait()</td>
<td>sigprocmask()</td>
</tr>
<tr>
<td>sem_unlink()</td>
<td>sigqueue()</td>
</tr>
<tr>
<td>sem_wait()</td>
<td>sigsetjmp()</td>
</tr>
<tr>
<td>setdomainname()</td>
<td>sigsetmask()</td>
</tr>
<tr>
<td>setegid()</td>
<td>sigsuspend()</td>
</tr>
<tr>
<td>seteuid()</td>
<td>sigtimedwait()</td>
</tr>
<tr>
<td>setgid()</td>
<td>sigunistd()</td>
</tr>
<tr>
<td>sethostname()</td>
<td>sigwait()</td>
</tr>
<tr>
<td>setitimer()</td>
<td>sigwaitinfo()</td>
</tr>
<tr>
<td>setjmp()</td>
<td>sleep()</td>
</tr>
<tr>
<td>setpgid()</td>
<td>slogb()</td>
</tr>
<tr>
<td>setpgrp()</td>
<td>slogf()</td>
</tr>
<tr>
<td>setprio()</td>
<td>slogi()</td>
</tr>
<tr>
<td>setregid()</td>
<td>snprintf()</td>
</tr>
<tr>
<td>setreuid()</td>
<td>sopenfd()</td>
</tr>
<tr>
<td>setrlimit()</td>
<td>spawn()</td>
</tr>
<tr>
<td>setrlimit64()</td>
<td>sprintf()</td>
</tr>
<tr>
<td>setsid()</td>
<td>srand()</td>
</tr>
<tr>
<td>settimeofday()</td>
<td>srand48()</td>
</tr>
<tr>
<td>setuid()</td>
<td>sscanf()</td>
</tr>
<tr>
<td>setutent()</td>
<td>stat()</td>
</tr>
<tr>
<td>shm_ctl()</td>
<td>stat64()</td>
</tr>
<tr>
<td>shm_ctl_special()</td>
<td>statvfs()</td>
</tr>
<tr>
<td>shm_open()</td>
<td>statvfs64()</td>
</tr>
<tr>
<td>shm_unlink()</td>
<td>straddstr()</td>
</tr>
<tr>
<td>sigaction()</td>
<td>strcasecmp()</td>
</tr>
</tbody>
</table>

Appendix: C • Summary of Safety Information November 2, 2006
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcat()</td>
<td>swprintf()</td>
</tr>
<tr>
<td>strchr()</td>
<td>swscanf()</td>
</tr>
<tr>
<td>strcmp()</td>
<td>symlink()</td>
</tr>
<tr>
<td>strcmpi()</td>
<td>sync()</td>
</tr>
<tr>
<td>strcoll()</td>
<td>sysconf()</td>
</tr>
<tr>
<td>strcpy()</td>
<td>sysmgr_reboot()</td>
</tr>
<tr>
<td>strerror()</td>
<td>tcdrain()</td>
</tr>
<tr>
<td>strftime()</td>
<td>tcdropline()</td>
</tr>
<tr>
<td>strftime()</td>
<td>tcflow()</td>
</tr>
<tr>
<td>stricmp()</td>
<td>tcflush()</td>
</tr>
<tr>
<td>strlen()</td>
<td>tcgetattr()</td>
</tr>
<tr>
<td>strlwr()</td>
<td>tcgetattrr()</td>
</tr>
<tr>
<td>strlwr()</td>
<td>tcgetpgrp()</td>
</tr>
<tr>
<td>strncasecmp()</td>
<td>tcgetsid()</td>
</tr>
<tr>
<td>strncat()</td>
<td>tcgetattr()</td>
</tr>
<tr>
<td>strncmp()</td>
<td>tcischars()</td>
</tr>
<tr>
<td>strncpy()</td>
<td>tcisstr()</td>
</tr>
<tr>
<td>strncmp()</td>
<td>tcsendbreak()</td>
</tr>
<tr>
<td>strncpy()</td>
<td>tcsetattr()</td>
</tr>
<tr>
<td>strnset()</td>
<td>tcspnb()</td>
</tr>
<tr>
<td>strpbrk()</td>
<td>tcspnbrr()</td>
</tr>
<tr>
<td>strrchr()</td>
<td>tcspn()</td>
</tr>
<tr>
<td>strrev()</td>
<td>tell()</td>
</tr>
<tr>
<td>strsep()</td>
<td>tell64()</td>
</tr>
<tr>
<td>strset()</td>
<td>time()</td>
</tr>
<tr>
<td>strsignal()</td>
<td>timer_create()</td>
</tr>
<tr>
<td>strspn()</td>
<td>timer_delete()</td>
</tr>
<tr>
<td>strstr()</td>
<td>timer_getexpstatus()</td>
</tr>
<tr>
<td>strtod()</td>
<td>timer_getoverrun()</td>
</tr>
<tr>
<td>strtoimax()</td>
<td>timer_gettime()</td>
</tr>
<tr>
<td>strtok_r()</td>
<td>timer_settime()</td>
</tr>
<tr>
<td>strtol()</td>
<td>timer_timeout()</td>
</tr>
<tr>
<td>strtooll()</td>
<td>timer_timeout_r()</td>
</tr>
<tr>
<td>strtoull()</td>
<td>times()</td>
</tr>
<tr>
<td>strtoq()</td>
<td>timespec2nsec()</td>
</tr>
<tr>
<td>strtoql()</td>
<td>tolower()</td>
</tr>
<tr>
<td>strupper()</td>
<td>toupper()</td>
</tr>
<tr>
<td>strxfrm()</td>
<td>towctrans()</td>
</tr>
<tr>
<td>swab()</td>
<td>towlower()</td>
</tr>
</tbody>
</table>
Signal handlers

`towupper()`  `wcsncpy()`
`truncatem()`  `wcsespn()`
`ttimezname`  `wcsxfmr()`
`ualarm()`  `wcsftime()`
`ultoa()`  `wcslen()`
`uhoai()`  `wcsncat()`
`umask()`  `wcsncmp()`
`uname()`  `wcsncpy()`
`unlink()`  `wcspbbr()`
`unsetenv()`  `wcsrchr()`
`usleep()`  `wcsrtombs()`
`utime()`  `wcssspn()`
`utimes()`  `wcsstr()`
`utmpname()`  `wctstod()`
`utoa()`  `wctstof()`
`va_arg()`  `wctstomax()`
`va_copy()`  `wctstok()`
`va_end()`  `wctstol()`
`va_start()`  `wctstold()`
`valloc()`  `wctstoll()`
`verr()`  `wctstombs()`
`verrx()`  `wctstoul()`
`vslogf()`  `wctstoul()`
`vwarn()`  `wctstounax()`
`vwarnx()`  `wctob()`
`wait()`  `wctomb()`
`wait3()`  `wctrans()`
`wait4()`  `wctype()`
`waitid()`  `wmemchr()`
`waitpid()`  `wmemcmp()`
`warn()`  `wmemcpy()`
`warnx()`  `wmemmove()`
`wcrtomb()`  `wmemset()`
`wcsct()`  `wordexp()`
`wcschr()`  `wordfree()`
`wcsncmp()`  `write()`
`wcsctoll()`  `writeblock()`
writev()

See the “Caveats” section for the following functions for more information:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort()</td>
<td>vsprintf()</td>
</tr>
<tr>
<td>modem_read()</td>
<td>vsscanf()</td>
</tr>
<tr>
<td>modem_script()</td>
<td>vswprintf()</td>
</tr>
<tr>
<td>vsnprintf()</td>
<td>vswscanf()</td>
</tr>
</tbody>
</table>

## Multithreaded programs

**CAUTION:** It isn’t safe to call these functions from a multithreaded program.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raccept()</td>
<td>getaddrinfo()</td>
</tr>
<tr>
<td>Rbind()</td>
<td>getc_unlocked()</td>
</tr>
<tr>
<td>Rconnect()</td>
<td>getc_unlocked()</td>
</tr>
<tr>
<td>Rgetsockname()</td>
<td>getenv()</td>
</tr>
<tr>
<td>Rlisten()</td>
<td>getgrent()</td>
</tr>
<tr>
<td>Rrcmd()</td>
<td>getgrgid()</td>
</tr>
<tr>
<td>Rselect()</td>
<td>getgrnam()</td>
</tr>
<tr>
<td>SOCKSinit()</td>
<td>gethostbyaddr()</td>
</tr>
<tr>
<td>bindresvport()</td>
<td>gethostbyname()</td>
</tr>
<tr>
<td>crypt()</td>
<td>gethostbyname2()</td>
</tr>
<tr>
<td>daemon()</td>
<td>gethostent()</td>
</tr>
<tr>
<td>drand48()</td>
<td>getlogin()</td>
</tr>
<tr>
<td>endgrent()</td>
<td>getnameinfo()</td>
</tr>
<tr>
<td>endhostent()</td>
<td>getnetbyaddr()</td>
</tr>
<tr>
<td>endnetent()</td>
<td>getnetbyname()</td>
</tr>
<tr>
<td>endprotoent()</td>
<td>getnetent()</td>
</tr>
<tr>
<td>endpwent()</td>
<td>getopt()</td>
</tr>
<tr>
<td>endservent()</td>
<td>getpass()</td>
</tr>
<tr>
<td>endspent()</td>
<td>getprotobynumber()</td>
</tr>
<tr>
<td>endutent()</td>
<td>getprotoent()</td>
</tr>
<tr>
<td>fgetspent()</td>
<td></td>
</tr>
</tbody>
</table>
Multithreaded programs

getpwent()  getpwnam()  getpwuid()  getservbyname()  getservbyport()  getservent()  gmtime()  herror()  inet_ntoa()  initgroups()  initstate()  input_line()  ioctl()  lcong48()  localtime()  lrand48()  lstat()  lstat64()  mrand48()  pclose()  popen()  putc_unlocked()  putchar_unlocked()  putenv()  putspent()  rand()  random()  rcmd()  readdir()  readdir()  res_init()  res_mkquery()  res_query()  res_querydomain()  res_search()

res_send()  ruserok()  seekdir()  setenv()  setgrent()  setgroups()  sethostent()  setkey()  setlogmask()  setprotoent()  setpwent()  setservent()  setspent()  setstate()  sigblock()  snmp_close()  snmp_free_pdu()  snmp_open()  snmp_pdu_create()  snmp_read()  snmp_select_info()  snmp_send()  snmp_timeout()  sockatmark()  srandom()  stmtok()  syslog()  telldir()  tempnam()  tyname()  vfork()  vsyslog()  wcscoll()
See the “Caveats” section for the following functions for more information:

- `ctermid()`
- `select()`
- `modem_read()`
- `tmpnam()`
- `modem_script()`
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.

**buildfile**

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 from 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 built-in 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.

ever-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 “Ti”. 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 `0xFFFFFFFF0` 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_stat() 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 FileSystem — 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 “Tsl”. 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
device 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) 1777
_rhosts 2690
/dev/name/global 1905
/dev/name/local 1905
/dev/zero 1705
/etc/autoconnect 1937
/etc/hosts 525, 873, 875, 877–880, 884, 885, 889, 1039
/etc/hosts.equiv 2690
/etc/networks 910, 912, 914, 1944, 2919
/etc/protocols 552, 936, 938, 940, 2153, 2928
/etc/resolv.conf 2550, 2552, 2555, 2558, 2561, 2564
/etc/services 556, 966, 968, 970, 2874, 2944
__cabsargs, __cabsfargs 258
__res_state 2548
__asyncmsg_connection_attr 183
__client_info 387
__clockadjust 347
__clockperiod 356
__cred_info 388
__CS_HOSTNAME 890, 2900
__CS_TIMEZONE 3593
__fdinfo 1273, 1278
__intrspin_t 1182, 1195, 1201
__io_connect 1212
__io_connect_ftype_reply 1219
__io_connect_link_reply 1221
__itimer 3546
__msg_info 1785, 1799
__NTO_SIDE_CHANNEL 382
__pccard_info 2031
__pulse 2410
__sched_info 2785
__server_info 396
__thread_attr 3484
__thread_local_storage 3487
__timer_info 3542
Index

1

8-bit characters, reading 1721
8086 mode, virtual 1208

A

abort() 119
abs() 121
absolute values
complex number 258
floating point 646
integer 121
long integer 1525
accept() 123, 3432, 3825
access() 126
ACCOUNTING 3645
acos(), acosf() 129
acosh(), acoshf() 131
ACTION 1041
adaptive partitions
averaging window size,
setting 357, 2748
bankruptcy, handling 2751
budgets
zero, preventing 2774–2776
zero, setting on
bankruptcy 2752
CPU time, getting 2763
creating 2754
IDs, finding for a
partition 2759
inheritance, preventing 303
locking 2774
maximum number of 2756
names
assigning 2755
maximum length 2755
parameters, setting 2761
processes
getting 2777
setting 2759
querying 2757
scheduler
controlling 2746
parameters, getting 2747
parameters, setting 2753
scheduling policies 2749
security, setting 2773, 2774
System 2754
threads
critical 2767, 2768, 3008
determining 2769
joining 2759, 2776
addresses
hosts
strings, converting
to/from 1119, 1122
IP
strings, converting
to/from 1097, 1099, 1114,
1116, 1119, 1122
IPv6 1127
link-local 1128
local network
IP addresses, converting
to/from 1103
IP addresses, extracting
from 1101
nodenames, translating 905
scoped 1128
site-local 1128
sockets 133, 765, 820, 825
addrinfo 133
  errors 820
  freeing 765
  getting 825
  advisory locks See files, locking
AF_INET 873, 876, 879, 910, 1039, 1066, 1118, 1122, 1416, 1944, 3432, 3599
AF_INET6 879, 1068, 1118, 1122, 1442
AF_LOCAL 3628
AF_UNSPEC 2654
AH (Authentication Header) 1423
AIMS (Auto Incrementing Mass Storage) 2022
aio_cancel() 135
aiocb 1548
aio_error() 137
aio_fsync() 139
aio_read() 141
aio_return() 143
aio_suspend() 145
aio_write() 147
alarm() 149
alarms, scheduling 149, 3596
aligned memory, allocating 1643, 2111
alignment, setting for a process 3161
alloca() 152
alphabetical, testing a character for 1454, 1485
alphanumeric, testing a character for 1452, 1483
alphasort() 155
__amblksiz 157, 1609, 2696, 3105
ANSI classification 104
APS_INIT_DATA() 2747
APS_PARTITION_NAME_LENGTH 2755
APS_QCRIT BILL AS_CRITICAL 2770
APS_QCRIT_PERM_CRITICAL 2770
APS_QCRIT_RUNNING_CRITICAL 2770
APS_SCHED_SEC_OFF 2774
APS_SYSTEM_PARTITION_NAME 2754
arccosines 129
architecture, instruction set 373
arcsines 164
arctangents 196, 198
__argc 159
argument lists, variable-length 3651, 3657, 3659, 3661
coercion 3652
arguments to main() 159, 160, 234
 parsing 916
__argv 160
arrays
  allocating 274, 2705
  quick-sorting 2451
  searching 250, 1536, 1583
ASCII, testing a character for 1456
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

asynchronous SNMP
  transactions  3123
  asyncmsg_channel_create()  172
  asyncmsg_channel_destroy()  174
  asyncmsg_connect_attach()  177
  asyncmsg_connectDetach()  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
  atoll()  209
  atol(), atoll()  211

atomic operations
  addition  213, 215
  bits
    clearing  217, 219
    setting  221, 223
    toggling  229, 231
  subtraction  225, 227
  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

attributes
  connection  183
  Authentication Header (AH)  1423
  authenticator()  3130
  Auto Incrementing Mass Storage
    (AIMS)  2022
  _auxv  234

B

background processes  415, 2136
  termination, notification  2138
bankruptcies
  handling 2751
barriers
  attributes 2206
  destroying 2210
  initializing 2214
  process-shared 2212, 2216
  destroying 2204
  initializing 2206
BARRIER_SERIALTHREAD 2208
basename() 235
bcmp() 238
bcopy() 240
_BEGIN_DECLS 113
Bessel functions
  first kind 1512, 1514, 1516
  second kind 3815, 3817, 3819
big endian
  __BIGENDIAN__ manifest 113
  little endian, converting
    to/from 539, 541, 543, 545, 547, 549, 3331
messages 1666
native format, converting
  to/from 527, 529, 531
ports
  reading from 1087, 1091
  writing to 1996, 2000
unaligned values
  accessing safely 3615, 3617, 3619
  writing safely 3609, 3611, 3613
  __BIGENDIAN__ 113
binary search 250
bind() 242, 3432, 3825
bindresvport() 245
BIOS (PCI), determining if present 2066
bits
  atomic operations
    clearing 217, 219
    setting 221, 223
    toggling 229, 231
  set, finding first 680
block buffering, setting for stream I/O 2877
block special devices
  reading 2489
  writing 3807
blocks
  allocating 3105
  reading from a file 2489
  system message log, writing to 3097
  writing to a file 3807
blocksize, filesystem 787, 3238
booting 3389
time since 3393
BOOT.TIME 1005, 3644
break condition, asserting 3435
break pointer
  advancing 2696
  increment 157
brk() 247
BRKINT 3456
bsearch() 250
BSS data 519, 522
_btext 253
btowc() 254
budgets
  setting 2761
  zero
  bankruptcy, setting on 2752
### Index

© 2006, QNX Software Systems GmbH & Co. KG.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>November 2, 2006</td>
</tr>
</tbody>
</table>

- Preventing: 2774–2776
  - Buffers: 2774–2776
    - Canonical input: 745, 2008
    - Freeing: 187
    - Locking: 1891
    - Raw input: 745, 2008
    - Stream I/O: 2875, 2962
      - Block: 2877
      - Flushing: 3416
      - Line: 2912
  - BUFSIZ: 2875
  - BULK_REQ_MSG: 3117
- Bus mastering, enabling: 2046
- Busy-waiting: 1920, 1923, 1926, 1928, 1930
- Bytes: 2875
  - Comparing: 238
  - Copying: 240, 1645, 1653
  - Overlapping objects: 1660
  - Multibyte character, number of bytes in: 1618, 1621
  - Reading: 2123, 2480
  - Setting: 1662
  - Writing: 2445, 3802
  - Zeroing: 256
- bzero(): 256
- Calendar times
  - Current: 3506
  - Local times, converting to/from: 1559, 1561, 1693
  - tm: 3564
- calloc(): 3131
- cancel(): 274
- Cancellation
  - Cleanup handlers: 2220, 2222
  - Points: 3487
  - Creating: 2406
  - State: 2365, 3480
  - Type: 2367, 3480
- Canonical input buffer: 745, 2008
- C_ANY: 2551, 2554, 2557, 2560
- Card Information Structure (CIS), reading: 2036
- CardBus cards: 2079
- cbrt(), cbrtf(): 276
- C_CHAOS: 2551, 2554, 2557, 2560
- ceil(), ceilf(): 278
- cfgetispeed(): 280
- cfgetospeed(): 283
- CFGFILE.Append: 287
- CFGFILE.Create: 287
- CFGFILE.Excl: 288
- CFGFILE.Rdonly: 287
- CFGFILE.Rdwr: 287
- CFGFILE.Trunc: 288
- CFGFILE.Wronly: 287
- cfgopen(): 287
- cfmakeraw(): 290
- cfreset(): 292
- cfsetispeed(): 294
- cfsetospeed(): 297
- ChannelCreate():
  - ChannelCreate.r(): 300

C

C++ programs
- End of C code: 113
- Start of C code: 113
- cabs(), cabsf(): 258
ChannelDestroy()

ChannelDestroy()

channels
creating 172, 300, 1904
destroying 174, 307, 1913
events, delivering 1788
flags 302
ID 301
messages
receiving 1818, 1830
replying 1834, 1837
sending 1842, 1847, 1855,
1859, 1863, 1867, 1871,
1875
pulses
receiving 1824, 1827
sending 1851
side channels 382
character device terminal drivers,
providing session support
to 2150
characters See also strings; wide
characters
8-bit, reading 1721
control
disabling 746, 2009
discarding on input 1721
default (signed or
unsigned) 113
devices
input stream, injecting 3427
size 3425, 3446
escape 1733
handling 2915
international See wide
characters
lowercase, converting to 1721,
3572
multibyte
bytes, counting 1618, 1621
wide characters, conversion
object 1626
wide characters, converting
to/from 1623, 1628,
1630, 1633, 3725, 3753,
3769, 3779
number waiting to be
read 3430
searching for 1647, 3252, 3296
sets, searching for 3262, 3294,
3306
special 1733
stdin, reading from 683, 836,
838
stdout, writing to 752, 2423,
2425
streams
pushing back 3624
reading from 681, 687, 832,
834
writing to 750, 2419, 2421
testing for
alphabetic 1454
alphanumeric 1452
ASCII 1456
ccontrol character 1460
decimal digit 1462
hexadecimal digit 1507
lowercase 1470
printable 1466, 1474
punctuation 1476
uppercase 1481
whitespace 1478
uppercase, converting to 3574,
3580
wide characters, converting
to/from 254, 3777
CHAR_MAX 1556
__CHAR_SIGNED__ 113
__CHAR_UNSIGNED__ 113
chdir() 310
chmod() 313, 1236
resource managers,
implementing in 1246,
1249
chown() 317, 1236
resource managers,
implementing in 1251,
1254
restricting use of 746, 2009
chroot() 320
C_HS 2551, 2554, 2557, 2560
CIDR (Classless Internet Domain
Routing) 1105, 1110
C_IN 2551, 2554, 2557, 2560
CIS (Card Information Structure),
reading 2036
classes
IP addresses 1106
PCI 2055
wide-character 3784
Classless Internet Domain Routing
(CIDR) 1105, 1110
clearenv() 326
clearerr() 329
cli 1185
CLK_TCK 3558
CLOCAL 3457
clock
adjusting 348
CPU time, getting ID of for a
thread 2267
cycles 350
getting 360
ID, getting 333
period, getting and setting 357
resolution, getting 335
setting 344, 360
ticks 3375
per seconds 3393
time, getting 337
clock_t 3375, 3557
clock() 331
ClockAdjust(), ClockAdjust_r() 348
ClockCycles() 350
clock_getcpu() 333
clock_gettime() 337
ClockId(), ClockId_r() 353
CLOCK_MONOTONIC 340, 3523,
3531, 3534, 3549
clock_nanosleep() 341
ClockPeriod(), ClockPeriod_r() 357
CLOCK_REALTIME 340, 344, 347,
356, 360, 2297, 2342, 2345,
2856, 3508, 3523, 3531,
3534, 3549
clock_settime() 344
CLOCK_SOFITIME 340, 3508,
3523, 3531, 3534, 3549
CLOCKS_PER_SEC 331
ClockTime(), ClockTime_r() 360
close() 363
resource managers, implementing in 1258, 1261

closedir() 365, 1981
closelog() 368
.cmsfd() 369
.cmdname() 370
cmsghdr 3629
code, portability 104
collating sequence, setting 2915
COLUMNS 326
commands
executing 3397
on a remote host 2469, 2660
options, parsing 916, 994
communications line
break condition, asserting 3435
disconnecting 3410
comparison
bytes 238, 1650, 1657
strings
case-insensitive 1657, 3247, 3256, 3274
case-sensitive 1650, 3254
locale’s collating sequence, using 3258
wide-character 3731, 3733, 3745, 3788
substrings
case-insensitive 3280, 3290
case-sensitive 3285
compiling with optimization 113
complementary error function 569
complex numbers, absolute value of 258
computer, rebooting 3389

condition variables
attributes
clock 2244, 2250
destroying 2242
initializing 2248
process-shared 2246, 2252
blocking on 2234, 2238
destroying 2227, 3352
initializing 2229, 3372
synchronization objects 3341, 3344
unblocking
all threads 2225
highest priority thread 2232
waiting on 2238
timed 2234
configuration files, opening 287, 649
configuration strings
.cs.hostname 890, 2900
.cs.timezone 3593
getting and setting 373
configuration values, getting 745, 1352, 2008, 3375
confstr() 373
connect functions (resource managers)
default values, setting 1280
open, default 1343
connect() 378, 3432, 3824
ConnectAttach(), ConnectAttach_r() 382
ConnectClientInfo(), ConnectClientInfo_r() 387
Index

ConnectDetach(),
  ConnectDetach_r()  303, 390
ConnectFlags(), ConnectFlags_r()  393
connection
  attributes
    structure  183
  buffers, attributes  179
connections
  allocating  191
  client, information about  387
detaching  181, 303, 390
dispatch interface,
  creating  1671
  flags, modifying  393
  ID  383
server, information about  395
shutting down
  full-duplex  2988
sockets
  accepting on  123, 2455
  initiating on  378, 1937, 2472
  listening on  1552, 2648
ConnectServerInfo(),
  ConnectServerInfo_r()  395
console I/O
  stderr  69, 3242
  stdin  69, 3243
  stdout  69, 3244
const  103
contention scope  2170, 2194
control characters  3458
disabling  746, 2009
discarding on input  1721
testing a character for  1460, 1487
controlling terminals
  making  3444
  path name  410
conventions
  typographical liii
copysign(), copysignf()  398
core files, maximum size  2938
cos(), cosf()  400
cosh(), coshf()  402
cosines  400
  hyperbolic  402
  inverse hyperbolic  131
CPU usage
  overall, getting  2765
  partition, getting for  2763
CREAD  3457
creat(), creat64()  404, 3605
critical threads
  determining if  2769
  marking as  2767, 3009
crypt()  408
CS5,...,CS8  3457
_CSArchitecture  373
_CS_ARCHITECTURE  373
_CS_DOMAIN  374
_CS_HOSTNAME  374
_CS_HW_PROVIDER  374
_CS_HW_SERIAL  374
CSIZE  3457
_CS_LIBPATH  374
_CS_MACHINE  374
_CS_PATH  374
_CS_RELEASE  374
_CS_RESOLVE  374
_CS_SRPC_DOMAIN  374
_CS_SYSNAME  374
_CS_TIMEZONE 374
CSTOPB 3457
_CS_VERSION 374
ctermid() 410
ctime(), ctime_r() 412
CTL_MAXNAME 3384
CTL_NET 3379, 3380
cube roots 276
current working directory 310, 840, 1017

d
daemon() 415
daemons
  SNMP (Simple Network Management Protocol),
  configuration file for 2485
  system 415, 2136
  termination, notification of 2138
data segment
  changing space allocated for 247
  end of 519, 522
  maximum size 2938
data server
  applications
deregistering 500
registering 506
variables
  creating 498
deleting 495
  flags, setting 502
  getting 504
  setting 508
data streams, flow control 3413
databases
  blocks
    reading 2489
    writing 3807
groups
  closing 523
  ID, getting information about 858, 860
  membership 1159
  name, getting information about 863, 865
  next entry, getting 855
  rewinding 2894
hosts
  closing 525
  entries, getting 873, 877, 879, 883, 885, 888
  errors 1030, 1036, 1045
  opening 2898
  structure 1039
network
  closing 551
  entries, getting 910, 912, 914
  opening 2919
  structure 1944
passwords
  closing 554
  encrypting 408, 2448
  entry, getting for a user 945, 948, 950, 953
  entry, getting next 942
  rewinding 2930
protocols
closing 552
entries, getting 936, 938, 940
opening 2928
protoent 2153
services
closing 556
entries, getting 966, 968, 970
entry structure 2874
opening 2944
shadow passwords
closing 557
entry, reading 690, 987, 992
entry, structure 2432
entry, writing 2432
rewinding 2952
system packet forwarding See ROUTE
datagrams 3140, 3141, 3599
date 3593
daylight 417, 3593
daylight saving time 417, 3593
_DCMD_ALL 427
_DCMD_ALL_GETFLAGS 1267
_DCMD_ALL_GETMOUNTFLAGS 1267
_DCMD_ALL_SETFLAGS 1267
_DCMD_BLK 427
_DCMD_CAM 427
_DCMD_CHR 427
_DCMD_FSYS 427
_DCMD_INPUT 427
_DCMD_IP 427
_DCMD_MEM 427
_DCMD_MISC 427
_DCMD_MIXER 427
_DCMD_NET 427
_DebugBreak() 418
decimal digit, testing a character for 1462, 1491
decimal-point character, setting 2915
delay() 423
delaytimer_max 3516
dev_t 2683
devctl() 426
DebugBreak() 418
DebugKDBreak() 420
DebugKDOOutput() 421
decimal digit, testing a character for 1462, 1491
decimal-point character, setting 2915
delay() 423
delaytimer_max 3516
dev_t 2683
devctl() 426
resource managers, implementing in 1267, 1271
_DEVCTL_DATA() 1269
DEVDIR_FROM 427
DEVDIR_NONE 427
DEVDIR_TO 427
DEVDIR_TOFROM 427
devices
block special
reading 2489
writing 3807
character
characters, injecting 3427
size 3425, 3446
classes of 2682
controlling 426, 1224
controlling device, making 3444
I/O memory, mapping 1710, 1897
mounts, autodetecting 1740
numbers
attaching 2681
detaching 2685
getting 2591
manipulating 3234
opening 1716
output, waiting for completion 3408
PCI
attaching 2041
configuration, reading 2068, 2071, 2074, 2077
configuration, writing 2081, 2084, 2086, 2088
detaching 2053
finding 2055, 2058
rescanning for 2079
reading 1722, 2492
script, running on 1726
writing 1733
devp-pccard server
arming 2023
attaching 2026
card insertion/removal, notification of 2023
CIS (Card Information Structure), reading 2036
detaching 2028
locking 2033
socket setup information 2030
unlocking 2037

_DEXTRA_FIRST() 2499
_DEXTRA_NEXT() 2499
_DEXTRA_VALID() 2499
D_FLAG_FILTER 437
D_FLAG_STAT 438
difftime() 435
digit, testing a character for
decimal 1462, 1491
hexadecimal 1505, 1507
_DIOF() 426
_DION() 426
_DIOT() 426
_DIOTF() 426
DIR 2498
dircntl() 437
direct memory access (DMA) channels, managing 2666
directories
access, checking 126, 516
base name 235
closing 365
controlling 437
creating 1679, 1685
current working 310, 840, 1017
for daemons 2136
deleting 2650
entries
duplicate, filtering 437
sorting 155
files, searching for 2013
hierarchy, walking 804, 1957
information, requesting 438
name 440
opening 1981
position
getting 3451
Index

setting 2822
reading 2498, 2502
rewinding 2638
root 320
scanning 2708
dirent 2498, 2502
dirname() 440
dispatch_t 453
dispatch interface See also resource managers
blocking 443
connections, creating 1671
context
allocating 447
freeing 450
endian, converting 1666
events
handling 459
last selected 2838
file descriptors
handle, attaching 2831
handle, detaching 2834
handles
creating 452
destroying 456
message handlers
attaching 1665
detaching 1674
names
attaching 1904
detaching 1913
server connections,
closing 1911
server connections,
opening 1915
path
attaching to 2568
detaching from 2587
pulse handlers
attaching 2413
detaching 2416
thread pool
attributes, changing 3461,
3474
creating 3463
destroying 3470
starting 3476
timeout, setting 463
unblocking 466
dispatch_block() 443
dispatch_context_alloc() 447
dispatch_context_free() 450
dispatch_create() 452
dispatch_destroy() 456
dispatch_handler() 459
dispatch_timeout() 463
dispatch_unblock() 466
div_t 468
div() 468
division
integer 468
long integer 1534
DL_info 470
dladdr() 471
dlclose() 473
DL_DEBUG 482
dlerror() 475
dlopen() 477
dlsym() 484
DMA channels, managing 2666
dn_comp() 487
dn_expand() 489
domains
errors 1030, 1036, 1045
Index

wide-character strings, converting to/from 3759

drand48() 491
drem(), dremf() 493
ds_clear() 495
ds_create() 498
ds_deregister() 500
ds_flags() 502
ds_get() 504
DS_PERM 497, 500, 502
ds_register() 506
ds_set() 508
_DTYPE_LSTAT 1361, 1408, 2499
_DTYPE_NONE 2499
_DTYPE_STAT 2499
dup() 510
dup2() 513
duplex connection, shutting down 2988
dynamically linked libraries addresses, translating 471
closing 473
debugging 482
ersrors 475
opening 477
symbol, getting address of 484

E

E2BIG 575
EACCES 575
eaccess() 516
EADDRINUSE 575
EADDRNOTAVAIL 575
EADV 575
eafnsupport 575, 3145
EAGAIN 575
EALREADY 575
EBADF 575, 2023, 2034, 2038
EBADFD 575
EBADFSYS 575
EBADMSG 575
EBADR 575
EBADRPC 575
EBADRQC 575
EBADSLT 575
EBFONT 575
EBUSY 575, 2034
ECANCELED 575
ECHILD 575
ECHO 3457
ECHOE 3457
ECHOK 3457
ECHONL 3458
ECHOS 575
ECHRNG 575
ECOMM 575
ECONNABORTED 575
ECONNREFUSED 575, 1552
ECONNRESET 575
ECTRLTERM 575
_edata 519
EDEADLK 575
EDEADLOCK 575
EDESTADDRREQ 575
EDESTADDRREQ 575
EDOM 575
EDQUOT 575
EEXIST 575
EFAULT 575
EFBIG 575
EHOSTDOWN 575
EHOSTUNREACH 575

3910

Index

November 2, 2006
EIDRM  575
EILSEQ  575
EINPROGRESS  575
EINTR  575
EINVAL  575
EIO  575
EISCONN 575, 3824
EISDIR  575
EL2HLT  575
EL2NSYNC 575
EL3HLT  575
EL3RST  575
ELIBACC  575
ELIBBAD  575
ELIBEXEC 575
ELIBMAX  575
ELIBSCN  575
ELNRNG  575
ELOOP  575
EMFILE  575
EMLINK  575
EMORE  575
EMPTY  3644
EMSGSIZE 575
EMULTIHOP 575
ENAMETOOLONG 575
Encapsulated Security Payload
(ESP)  1423
encrypt()  520
encryption
  key, setting  2910
  passwords  408, 2448
  strings  520
_end  522
end-of-file
  files  562
  streams  674
clearing  329
_END_DECLS  113
endgrent()  523
endhostent()  525
big
  _BIGENDIAN_... manifest  113
  little endian, converting
    to/from  539, 541, 543,
      545, 547, 549, 3331
  native format, converting
    to/from  527, 529, 531
  unaligned values, accessing
    safely  3615, 3617, 3619
  unaligned values, writing
    safely  3609, 3611, 3613
little
  big endian, converting
    to/from  539, 541, 543,
      545, 547, 549, 3331
  _LITTLE_ENDIAN_... manifest  113
  native format, converting
    to/from  533, 535, 537
  unaligned values, accessing
    safely  3615, 3617, 3619
  unaligned values, writing
    safely  3609, 3611, 3613
ports
  reading from  1087, 1091
  writing to  1996, 2000
ENDIAN_BE16()  527
ENDIAN_BE32()  529
ENDIAN_BE64()  531
Index

ENIAND\_LE16() 533
ENIAND\_LE32() 535
ENIAND\_LE64() 537
ENIAND\_RET16() 539
ENIAND\_RET32() 541
ENIAND\_RET64() 543
ENIAND\_SWAP16() 545
ENIAND\_SWAP32() 547
ENIAND\_SWAP64() 549
endnetent() 551
endprotoent() 552
endpwent() 554
endservent() 556
endspent() 557
endutent() 559
ENETDOWN 575
ENETRESET 575
ENETUNREACH 575
ENFILE 575
ENOANO 575
ENOBUFS 575, 2655
ENOCI 575
ENODATA 575
ENODEV 575, 2032, 2034, 2038
ENOENT 575
ENOEEXEC 575
ENOLCK 575
ENOLIC 575
ENOLINK 575
ENOMEM 575
ENOMSG 575
ENONDND 575
ENONET 575
ENOPKG 575
ENOPROTOOOP 575
ENOREMOTE 575
ENOSPC 575
ENOSR 575
ENOSTR 575
ENOSYS 575
ENOTBLK 575
ENOTCONN 575
ENOTDIR 575
ENOTEMPTY 575
ENOTSOCK 575
ENOTSUP 575
ENOTTY 575
ENOTUNIQ 575
ENTER 1041
ENTRY 1041
env 606, 629, 2885, 3169, 3179, 3196, 3206
environ 561, 1604, 2886
environment
 restoring 1578, 3015
 saving 2907, 3061
environment variables
 COLUMNS 326
defining 561, 606, 629, 2427, 2885, 3158, 3174, 3184, 3189, 3201, 3210
deleting 326, 2427, 2885, 3158, 3174, 3184, 3189, 3201, 3210, 3634
DL\_DEBUG 482
files, searching for 2817
getting 849
HOSTALIASES 880
HOSTNAME 890, 2900
INCLUDE 2817
LD\_LIBRARY\_PATH 483
LIB 2817
LINES 326

3912 Index November 2, 2006
LOCALDOMAIN  2548, 2552, 2556, 2558, 2561, 2564
PATH  57, 326, 605, 628, 2817, 3178, 3183, 3188, 3205, 3209
pointer to  561
SHELL  326, 2105, 3398
SNMPCONFIGFILE  2487
TERM  326
TERMINFO  326
TZ  326, 3593
ENXIO  575
EOF  3668
eof()  562
EOK  575
EOPNOTSUPP  575, 3145
EOVERFLOW  575
EPERM  575
EPFNOSUPPORT  575
EPIPE  575
EPROCUNA VAIL  575
EPROGMISMA TCH  575
EPROGUNA VAIL  575
EPROTO  575
EPROTONOSUPPORT  575, 3145
EPROTOTYPE  575
erand48()  565
ERANGE  575
EREMCHG  575
EREMOTE  575
ERESTART  575, 1796
erf(), erff()  567
erfc()  569
erfcf()  569
EROF S  575
ERPCMISMATCH  575
err(), errx()  571
errno  574
threads  575, 3487
error function  567
complementary  569
errors  See also signals
command-line options, printing
for  917
end-of-file  674
errno global variable  574
hosts  1030, 1036, 1045
message-passing  1796
messages
for an error code  2092, 3266
formatted  571, 3665
regular expressions  2533
resolver  1030, 1036, 1045
signals, raising  2457
socket address
information  820
stderr  69, 571, 2092, 3242, 3665
stream I/O
clearing  329
testing for  676
escape characters  1733
ESHUTDOWN  575
ESOCKTNOSUPPORT  575
ESP (Encapsulated Security Payload)  1423
ESPIPE  575
ESRCH  575
ESRMNT  575
ESRVRFAULT  575
ESTALE  575
ESTRPIPE  575
_estext  583
ETIME 575
ETIMEDOUT 575
ETOOMANYREFS 575
ETXTBSY 575
EUNATCH 575
EUSERS 575
events
    blocking while waiting for 443
    channels, delivering  through 1788
    checking validity of 1878
    handling 459
interrupts
    attaching 1176
    detaching 1182
    last selected 2838
sigevent 3004
    type, determining with
      SIGEV_GET_TYPE() 3005
system
    notification of 2139
    triggering 2143
    tracing 3582
unblocking 466
EWOULDBLOCK 575, 983
exceptional conditions, file
    descriptors with 2665, 2825
exceptions, floating-point
    mask 734
    registers 736
exclusive locks 707
EXDEV 575
exec* family of functions 49, 57, 1981
exect() 584
execl() 592
execlp() 599
execlpe() 605
executable files
    base name 2152
    file descriptor 369
    full path 370
    mounted filesystem, preventing
      from loading on 1739
execv() 609
execve() 615
execvp() 621
execvepe() 628
EXFULL 575
_exit() 632
exit() 635
exp(), expf() 638
expm1(), expm1f() 640
exponentials, floating point 638,
    640, 773, 1532
exponents,
    radix-independent 1573,
    2699, 2702
export 606, 629, 2885, 3169,
    3179, 3196, 3206
F
fabs(), fabsf() 646
F_ALLOCSP 660, 1371
fcfgopen() 649
fchmod() 650
fchown() 653
FCHR_MAX 2942
fclose() 656
fcloseall() 658
fcntl() 660
FD DUPFD 510, 513
resource managers, implementing in 1291
FD See file descriptors
fdatasync() 669
FD CLOEXEC 662, 2980, 3157, 3188
FD CLR() 2826
FDINFO() 2826
resource managers, implementing in 1274, 1276
_FDINFO_FLAG_LOCALPATH 1226, 1277
FD ISSET() 2826
fdopen() 671
FD_SET() 2826
FD_SETSIZE 2826
FD_ZERO() 2826
feof() 674
ferror() 676
fflush() 678
F_FREESP 661, 1371
ffs() 680
fgetc() 681
fgetchar() 683
F GETFD 661
F_GETFL 661
F_GETLK 661
fgetpos() 685
fgets() 687, 964
fgetspent() 690
fgetwc() 693
fgetws() 695
FIFO scheduling 2192
FIFOs creating 1682, 1685
reading from 2123, 2480, 2508
unnamed (pipes)
closing 2090
creating 2094
opening 2104
_FILE_ 168
file descriptors closing in a child process 3157, 3188
connection IDs as 383
creating 1974, 3150
duplicating 510, 513, 661, 1984, 3155
exceptional conditions 2665, 2825
full path 3588, 3590
handle attaching 2831
detaching 2834
maximum number of 2938
polling 2097
properties 1464
ready for reading or writing 2665, 2825
selecting 2665, 2825
sets of, manipulating 2826
stderr 3242
stdin 3243
stdout 3244
streams, associating with 671, 698
table size, getting 845
terminal, testing for association with 1458
Index

fileno()  69, 698
files
access times  3641
disabling logging of on mounted filesystems 1739
resource managers, implementing in 1386, 1398, 1401
access, checking  126, 516
base name  235
closing  363
configurable limits  745, 1352, 2008
configuration, opening  287, 649
controlling  660
core, maximum size  2938
creating  722, 1974, 3150
low-level  404
not allowing on mounted filesystems 1739
resource manager  1214, 1338
resource manager  1214, 1338
deleting  2542, 3631
device parameters, manipulating  1224
directory name  440
executable
base name  2152
file descriptor  369
full path  370
extending  323, 660
flushing  678
all  713
information, getting  782, 1590, 3230
input, formatted  775, 818, 3672, 3677
link count, maximum  745, 2008
linking to  703, 1543, 2016, 2018, 2505, 3337
locking  661, 706, 1564
by a thread  709
nonblocking  801
modification times  808, 3638, 3641
resource managers, implementing in 1386, 1398, 1401
names
full path  3588, 3590
matching  719
maximum length  745, 2009
opening  404, 722, 1974, 3150
resource manager  1338
output, formatted  748, 813, 3669, 3675
ownership, changing  317, 653, 1527
pathnames matching a pattern  1020, 1024
_PATH_UTMP  559
_PATH_UTMP  1003, 1006, 1008, 2437, 2960, 3646
permissions
changing  313, 650
daemons  2136
on creation  3604
restricting the changing of  746, 2009
position
getting  685
setting 778, 780, 1587, 3448
private access 1984
processes, maximum files per 3376
reading 1164, 2123, 2480
blocks 2489
characters, number waiting to be read 3430
checking 2474
iov
2508
renaming 2545
reopening 769
rewinding 2635
scanning directories for 2708
searching
environment variables 2817
list of directories 2013
seeking 778
shared access 3150, 3155
size, changing 323
size, maximum 2938
SNMP configuration 2485
status flags 661
status-change times
resource managers, implementing in 1386, 1398, 1401
synchronizing 669, 790
temporary
creating 3566
creating and opening 1689
name, generating 1691, 3453, 3569
tree, walking 804, 1957
truncating 323, 661, 798, 1597, 3585
unlinking 2542, 3631
unlocking 662, 706, 806, 1564
writing 2445, 3802
blocks 3807
characters 750, 752
iov_t 3810
strings 754
wide characters 756
wide-character strings 758
filesystems
free space 787
information, getting 786, 3238
mounting 1740
options, parsing 1742
read-only 1739
synchronizing 3340
requesting 2143
unmounting 1740, 3607
FIND 1041
finite number, determining if 701
finite(), finitef() 701
first-in first-out scheduling 2192
flink() 703
floating point
absolute value 258, 646
arccosines 129
arcsines 164
arctangents 196, 198
Bessel functions 1512, 1514, 1516, 3815, 3817, 3819
complementary error function 569
cosines 400
cube roots 276
error function 567
exceptions, signal for 3025
exponentials 638, 640
Index

© 2006, QNX Software Systems GmbH & Co. KG.

exponents,
radix-independent 2699, 2702
finite, determining if 701
fractional part 1736
gamma functions 822, 1539
hyperbolic cosines 402
hyperbolic sines 3079
hyperbolic tangents 3406
hypotenuse, length of 1060
infinite, determining if 1468
input, formatted 2709
integral logarithms 1081
integral part 1736
integral power of 2 773, 1532
inverse hyperbolic cosines 131
inverse hyperbolic sines 166
inverse hyperbolic tangents 200
logarithms 1567, 1569, 1571
modular arithmetic 715
next representable 1954
normalized fractions 773
not a number, determining if 1472
powers 2121
printing 2126
radix-independent exponents 1573, 2699, 2702
remainders 493, 2540
residue 715
rounding 278, 711, 2645
settings
exception mask 734
exception registers 736
precision 739
rounding 742
sign, copying 398
significant bits 3048
sines 3077
square roots 3217
tangents 3404
wide-character strings,
converting to/from 3759

flock 663
flock() 706
flockfile() 709
floor(), floorf() 711
flow control 3413
flushall() 713
fmod(), fmodf() 715
fnmatch() 719
FN M_PATHNAME 718
FN M_PERIOD 718
FN M QUOTE 718
F_OK 126, 516
fopen() 722
fork() 727, 1981
forkpty() 731
fpathconf() 745
_FP EXC_DENORMAL 733, 736
_FP EXC_DIVZERO 733, 736
_fp_exception_mask() 734
_fp_exception_value() 736
_FP EXC_INEXACT 733, 736
_FP EXC_INVALID 733, 736
_FP EXC_OVERFLOW 733, 736
_FP EXC_UNDERFLOW 733, 736
fpos_t 685, 780
_FP_PREC_DOUBLE 739
_FP_PREC_DOUBLE_EXTENDED 739
_FP_PREC_EXTENDED 739
_FP_PREC_FLOAT 739

3918 Index

November 2, 2006
fp_precision() 739
fprintf() 748
fp_rounding() 742
_FP_ROUND NEAREST 742
_FP_ROUND NEGATIVE 742
_FP_ROUND POSITIVE 742
_FP_ROUND ZERO 742
fputc() 750
fputchar() 752
 fputs() 754
fputwc() 756
fputws() 758
FQNN (Fully Qualified Node Name) 1946
FQPN (Fully Qualified Path Name) 1945
fractional part of a floating-point number 1736
F_RDLCNK 662
fread() 760
free memory, amount of 3236
free space, filesystem 787, 3239
free() 763
freeaddrinfo() 765
freeaddrinfo() 767
freopen() 69, 769
frex() , frexp() 773
fscanf() 775
fseek() 778, 2635
F_SETFD 662
F_SETFL 662
F_SETLK 662
F_SETLKW 663
fsetpos() 780
fstat(), fstat64() 782
fstatvfs(), fstatvfs64() 786
fsync() 790
f tell() 792
ftime() 795
fruncate(), fruncate64() 798
ftrylockfile() 801
ftw() 804
FTW_D 804, 1958
FTW_DNR 804, 1958
FTW_F 804, 1958
FTW_NS 804, 1959
_FTYPE_ANY 1213, 1219, 1221, 2566
_FTYPE_LNK 1213, 1219, 1221, 2566
_FTYPE_MNT 1213, 1219, 1221, 2566
_FTYPE_MQ 1213, 1219, 1221, 2566
_FTYPE_PIPE 1213, 1219, 1221, 2566
_FTYPE_SEM 1213, 1219, 1222, 2567
_FTYPE_SHM 1213, 1219, 1222, 2567
_FTYPE_SOCK 1213, 1219, 1222, 2567
_FTYPE_SOCK 1213, 1219, 1222, 2567
_FTYPE_SYMLNK 1213, 1219, 1222, 2567
full-duplex connection, shutting down 2988
Fully Qualified Node Name (FQNN) 1946
Fully Qualified Path Name (FQPN) 1945
function
classification 104
safety 108
F_UNLCK 662

November 2, 2006
Index

funlockfile() 806
futime() 808
fwrite() 811
fwprintf() 813
fwrite() 815
F_WRLCK 662
fwscanf() 818

gai_strerror() 820
gamma functions 822, 1539
gamma(), gamma_r(), gammaf(), gammaf_r() 822
General Purpose Interface Bus (GPIB) 2022
getaddrinfo() 825
cgetc() 832
getchar() 836
getchar_unlocked() 838
getcwd() 840
get_device_command() 427
get_device_direction() 427
getdomainname() 843
getdtablesize() 845
getegid() 847
getenv() 606, 629, 849, 3169, 3179, 3196, 3206
geteuid() 851
getgid() 853
getgrent() 855
getgrgid() 858
getgrgid_r() 860
getgrnam() 863
getgrnam_r() 865
getgrouplist() 869
getgroups() 871
gethostbyaddr() 873
gethostbyaddr_r() 877
gethostbyname(), gethostbyname2() 879
gethostbyname_r() 883
gethostent() 885
gethostent_r() 888
gethostname() 890
getifaddrs() 892
GETIOVBASE() 894
GETIOVLEN() 896
getitimer() 898
getlogin() 901
getlogin_r() 903
getnameinfo() 905
getnetbyaddr() 910
getnetbyname() 912
getnetent() 914
GETNEXT_REQ_MSG 3117
getopt() 916
getpass() 922
getpeername() 924
getpgid() 926
getpgrp() 928
getpid() 930
getppid() 932
getprio() 934
getprotobyname() 936
getprotobynumber() 938
getprotoent() 940
getpwnam() 942
getpwnam_r() 945
getpwuid() 950
getpwuid_r() 953
GET_REQ_MSG 3117
getrlimit(), getrlimit64() 956
GET_RSP_MSG 3117
getusage() 959
gets() 964
getservbyname() 966
getservbyport() 968
getservent() 970
getsid() 972
getsockname() 974, 3825
getsockopt() 977
getspent() 987
getspmnam(), getspmnam_r() 992
getsubopt() 994
gmtimeofday() 999
getuid() 1001
getutent() 1003
getutid() 1005
getutline() 1008
gew() 1011
getswct() 1013
getswchar() 1015
getwd() 1017
glob_t 1019
glob() 1020
global variables
  __ablksize 157
  __argc 159
  __argv 160
  __auxv 234
  __btext 253
daylight 417
  __edata 519
  __end 522
  __errno 574
  __etext 583
  optarg 916
  opterr 917
  optind 916
  optopt 917
  __progname 2152
  __stderr 3242
  __stdin 3243
  __stdout 3244
  __sys_errlist 575
  __sys_nerr 575
  __sys_nsig 3024
  __sys_page_ptr 3396
  __sys_siglist 3024
timezone 3563
  tzname 3592
globfree() 1024
gmtime() 1026
gmtime_r() 1028
GPIB (General Purpose Interface Bus) 2022
greater of two numbers 1616
groups See also process groups
  access list
    getting 869
    initializing 1159
database
  closing 523
  membership 1159
  next entry, getting 855
  rewinding 2894
IDs
  effective 847, 2882, 2891, 2931
  information about 858, 860
  process 928
  real 853, 2891, 2931
  saved 2891
name, getting information about 863, 865
set-group ID 3376
supplementary IDs 871, 2896
maximum per process 3375
guard area 2162, 2185
guardian process, specifying 2145

network-byte order, converting to/from 1047, 1049, 1967, 1969

H

HATELYASES 880
hostent 1039
HOSTNAME 890, 2900
HOST_NOT_FOUND 1030, 1037
hosts addresses
strings, converting to/from 1119, 1122
database closing 525
entries, getting 873, 877, 879, 883, 885, 888
errors 1030, 1036, 1045
opening 2898
structure 1039
domain name
getting 374, 843
setting 2880
names 890, 2900
getting 374
valid characters 374
remote identity, checking 2690
hot swapping 2079
hsearch() 1041
hstrerror() 1045
htonl() 1047
htons() 1049
HUPCL 3457
hwi_find_item() 1052
hwi_find_tag() 1055
HWI_NULL_OFF 1052
hwi_off2tag() 1056
hwi_tag2off() 1058

H

hardware
information in system
page 1052, 1055, 1056, 1058
manufacturer, getting 374
serial number, getting 374
type 374, 3621
hardware interrupts See interrupts
hash table
creating 1032
destroying 1034
searching 1041
hcreate() 1032
hdestroy() 1034
h_errno 874, 880, 1030, 1036, 2555, 2561
herror() 1036
hexadecimal numbers
digit, testing a character for 1505, 1507
strings, converting to/from 207
holding a process for debugging 3162
host-byte order
hyperbolic functions  
See also  
trigonometry  
hyperbolic cosine 402  
hyperbolic sine 3079  
hyperbolic tangent 3406  
inverse hyperbolic cosine 131  
inverse hyperbolic sine 166  
inverse hyperbolic tangent 200  
hypot(), hypotf() 1060  
hypotenuse, length of 1060  

I  

I/O  
buffers, flushing 3416  
configuration files, opening 287  
end-of-file, checking for 562  
FIFOs, creating 1682, 1685  
file descriptors  
  duplicating 510, 513, 661, 1984, 3155  
  selecting 2665, 2825  
file-mode creation mask 3604  
daemons 2136  
files  
  closing 363  
  controlling 660  
  extending 660  
  information, getting 782  
  linking to 1543, 2016, 2018, 2505, 3337  
  locking 661, 1564  
  names, matching 719  
  opening 404, 1974, 3150  
  reading 2123, 2480, 2489  
  status flags 661  
  truncating 661  
  unlocking 662, 706, 1564  
  writing 2445, 3802, 3807  
filesystems  
  information, getting 786, 3238  

iov_t  
  reading 2508  
  writing 3810  
ports, managing 2666  
privileges, requesting 1168, 1177, 1182, 1195, 1198, 1201, 1203, 3492  
requests, initiating list of 1548  

I/O functions (resource managers)  
chmod 1246  
  default 1249  
chown 1251  
  default 1254  
close  
  default 1258, 1261  
default values, setting 1280  
devctl 1267  
  default 1271  
fdinfo 1274  
  default 1276  
link 1284  
lock  
  default 1291  
lseek 1298  
  default 1301  
mknod 1304  
mmap 1306  
  default 1310  
openfd 1346
Index

<table>
<thead>
<tr>
<th>Function/Variable</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>default 1350</td>
<td></td>
</tr>
<tr>
<td>pathconf 1352</td>
<td></td>
</tr>
<tr>
<td>default 1355</td>
<td></td>
</tr>
<tr>
<td>read 1360</td>
<td></td>
</tr>
<tr>
<td>default 1357</td>
<td></td>
</tr>
<tr>
<td>readlink 1363</td>
<td></td>
</tr>
<tr>
<td>space 1371</td>
<td></td>
</tr>
<tr>
<td>stat 1374</td>
<td></td>
</tr>
<tr>
<td>default 1376</td>
<td></td>
</tr>
<tr>
<td>sync 1383</td>
<td></td>
</tr>
<tr>
<td>default 1381</td>
<td></td>
</tr>
<tr>
<td>utime 1398</td>
<td></td>
</tr>
<tr>
<td>default 1401</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td></td>
</tr>
<tr>
<td>default 1404</td>
<td></td>
</tr>
<tr>
<td>I/O vector</td>
<td></td>
</tr>
<tr>
<td>base, getting</td>
<td>894</td>
</tr>
<tr>
<td>fields, filling</td>
<td>2902</td>
</tr>
<tr>
<td>length, getting</td>
<td>896</td>
</tr>
<tr>
<td>reading from a file</td>
<td>2508</td>
</tr>
<tr>
<td>writing to a file</td>
<td>3810</td>
</tr>
<tr>
<td>ICRNL 3456</td>
<td></td>
</tr>
<tr>
<td>IEXTEN 3458</td>
<td></td>
</tr>
<tr>
<td>if _msghdr 2656</td>
<td></td>
</tr>
<tr>
<td>ifaddr 1079</td>
<td></td>
</tr>
<tr>
<td>if_freenamexindex() 1071</td>
<td></td>
</tr>
<tr>
<td>if_indextoname() 1073</td>
<td></td>
</tr>
<tr>
<td>if_nameindex() 1075</td>
<td></td>
</tr>
<tr>
<td>if_nametoinx() 1077</td>
<td></td>
</tr>
<tr>
<td>INAMDSIZE 1073</td>
<td></td>
</tr>
<tr>
<td>IGNBRK 3456</td>
<td></td>
</tr>
<tr>
<td>IGNCR 3456</td>
<td></td>
</tr>
<tr>
<td>IGNPAR 3456</td>
<td></td>
</tr>
<tr>
<td>IHFLOW 3457</td>
<td></td>
</tr>
<tr>
<td>ilogb(), ilogbf() 1081</td>
<td></td>
</tr>
<tr>
<td>in16() 1087</td>
<td></td>
</tr>
<tr>
<td>in16s() 1089</td>
<td></td>
</tr>
<tr>
<td>in32() 1091</td>
<td></td>
</tr>
<tr>
<td>in32s() 1093</td>
<td></td>
</tr>
<tr>
<td>in6_pktinfo 1446</td>
<td></td>
</tr>
<tr>
<td>in8() 1083</td>
<td></td>
</tr>
<tr>
<td>in8s() 1085</td>
<td></td>
</tr>
<tr>
<td>INADDR_ANY 1418, 3432</td>
<td></td>
</tr>
<tr>
<td>INADDR_NONE 1097, 1098, 1112</td>
<td></td>
</tr>
<tr>
<td>inetb16() 1087</td>
<td></td>
</tr>
<tr>
<td>inetb32() 1091</td>
<td></td>
</tr>
<tr>
<td>INCLUDE 2817</td>
<td></td>
</tr>
<tr>
<td>incoming connections, listening for 1552, 2648</td>
<td></td>
</tr>
<tr>
<td>index() 1095</td>
<td></td>
</tr>
<tr>
<td>INET6 (Internet protocol v6 family) 1127</td>
<td></td>
</tr>
<tr>
<td>INET6_ADDRSTRLEN 1118</td>
<td></td>
</tr>
<tr>
<td>inet6_option_alloc() 1131, 1133, 1135, 1137, 1139, 1141</td>
<td></td>
</tr>
<tr>
<td>inet6_option_append() 1131, 1133, 1135, 1137, 1139, 1141</td>
<td></td>
</tr>
<tr>
<td>inet6_option_find() 1131, 1133, 1135, 1137, 1139, 1141</td>
<td></td>
</tr>
</tbody>
</table>
inet6_option_init() 1131, 1133, 1135, 1137, 1139, 1141
inet6_option_next() 1131, 1133, 1135, 1137, 1139, 1141
inet6_option_space() 1131, 1133, 1135, 1137, 1139, 1141
inet6_rthdr_add() 1143
inet6_rthdr_getaddr() 1145
inet6_rthdr_getflags() 1147
inet6_rthdr_init() 1149
inet6_rthdr_lasthop() 1151
inet6_rthdr_reverse() 1153
inet6_rthdr_segments() 1155
inet6_rthdr_space() 1157
inet_addr() 1097
INET_ADDRSTRLEN 1118
inet_aton() 1099
inet_lnaof() 1101
inet_makeaddr() 1103
inet_net_ntop() 1105
inet_netoff() 1108
inet_netpton() 1110
inet_network() 1112
inet_ntoa() 1114
inet_ntoa_r() 1116
inet_ntop() 1119
inet_pton() 1122
infinite number, determining
    if 1468
INFORM_REQ_MSG 3117
inheritance 3161
initgroups() 1159
INIT_PROCESS 1005, 3644
initstate() 1161
INLCR 3456
inle16() 1087
inle32() 1091

inetnodes
    getting 2591
    number of 787, 3239
INPCK 3456
input, formatted 775, 818, 2709, 3672, 3677, 3681, 3705, 3813
input_line() 1164
input_line_max 1165
instruction set architecture 373
instrumented kernel 1193, 3582
_INT_BITS_ 113
integers
    absolute value 121
    atomic operations
        addition 213, 215
        subtraction 225, 227
        division 468
        pseudo-random numbers 2460, 2462
    quotient 468
    remainder 468
    rounding 278, 711
    size of 113
    system message log, writing
to 3103
integral logarithms 1081
integral part of a floating-point
number 1736
integral power of 2 773, 1532
Intel 80x86-specific interrupts 1208
interfaces
    index, mapping to name 1073
    list of, freeing 1071
    list of, getting 1075
    name, mapping to index 1077
international characters  See wide characters
Internet Control Message Protocol  See ICMP
Internet domain
  errors  1030, 1036, 1045
  name servers
    initializing  2548
    querying  2552, 2555, 2558, 2561, 2563
names
  compressing  487
  expanding  489
Internet Protocol  See IP
InterruptAttach(),
  InterruptAttach_r()  1168
InterruptAttachEvent(),
  InterruptAttachEvent_r()  1176
InterruptDetach(),
  InterruptDetach_r()  1182
InterruptDisable()  1185
InterruptEnable()  1187
InterruptHookIdle()  1189
InterruptHookTrace()  1193
InterruptLock()  1195
InterruptMask()  1198
interrupts
  classes  1168
  disabling  1185
  enabling  1187
  events
    attaching  1176
    detaching  1182
  handlers
    attaching  1168
    detaching  1182
  disabling hardware
    interrupts  1185
    guidelines for writing  1171
    idle, attaching  1189
    locking  1195, 1201
    stack size  1171
    Intel 80x86-specific  1208
    level-sensitive  1199
    masking  1198, 1203
    PCI
      mapping  2063
      routing information  2060
      requests, managing  2666
      waiting for  1205
InterruptUnlock()  1201
InterruptUnmask()  1203
InterruptWait(), InterruptWait_r()  1205
_intr_v86()  1208
inverse hyperbolic cosines  131
inverse hyperbolic sines  166
inverse hyperbolic tangents  200
io_chmod_t  1247
io_chown_t  1252
io_close_t  1259
io_devctl_t  1268
io_fdinfo_t  1277
io_link_t  1284
io_lseek_t  1299
io_mknod_t  1304
io_mmap_t  1307
io_notify_t  1314
io_open_t  1339
io_openfd_t  1347
io_pathconf_t  1353
io_read_t  1360
io_readlink_t  1364
io_rename_t 1367
io_space_t 1371
io_stat_t 1377
io_sync_t 1383
io_unlink_t 1394
io_utime_t 1399
io_write_t 1407
IO_CHMOD 1246, 1249
IO_CHOWN 1251, 1254
IO_CLOSE 1258, 1261
IO_COMBINE_FLAG 1248, 1252, 1259, 1269, 1277, 1299, 1307, 1315, 1347, 1353, 1361, 1371, 1377, 1384, 1399, 1408
IO_CONNECT 1212
IO_CONNECT 1343
IO_CONNECT_COMBINE 1212
IO_CONNECT_COMBINE_CLOSE 1212
IO_CONNECT_EFLAG_DIR 1217, 1222
IO_CONNECT_EFLAG_DOT 1217, 1222
IO_CONNECT_EXTRA_LINK 1217
IO_CONNECT_EXTRA_MOUNT 1218
IO_CONNECT_EXTRA_MOUNT_OCB 1218
IO_CONNECT_EXTRA_MQUEUE 1217
IO_CONNECT_EXTRA_NONE 1217
IO_CONNECT_EXTRA_PHOTON 1217
IO_CONNECT_EXTRA_PROC_SYMLINK 1218
IO_CONNECT_EXTRA_RENAME 1218
IO_CONNECT_EXTRA_RESMGR_LINK 1218
IO_CONNECT_EXTRA_SEM 1218
IO_CONNECT_EXTRA_SOCKET 1217
IO_CONNECT_EXTRA_SYMLINK 1217
IO_CONNECT_LINK 1213
IO_CONNECT_MKNOD 1213
IO_CONNECT_MOUNT 1213
IO_CONNECT_OPEN 1212
IO_CONNECT_READLINK 1213
IO_CONNECT_RENAME 1213, 1367
IO_CONNECT_RET_LINK 1338
IO_CONNECT_RSVD_UNBLOCK 1213
IO_CONNECT_UNLINK 1212
ioctl() 1224
IO_DEVCTL 1267, 1271
IOFBUF 2962
IO_FINFO 1274, 1276
iofdinfo() 1226
IO_FLAG_RD 1379
IO_FLAG_WR 1379
iofunc_attr_t 1233
iofunc_notify_t 1316
iofunc_ocb_t 1335
IOFUNCATTRTIME 1234, 1386, 1402
IOFUNCATTRCTIME 1234, 1386, 1402
IOFUNCATTRDIRTY_MODE 1234, 1402
IOFUNCATTRDIRTY_MTIME 1234
IOFUNCATTRDIRTY_NLINK 1234
IOFUNCATTRDIRTY_OWNER 1234
IOFUNCATTRDIRTY_RDEV 1234
IOFUNCATTRDIRTY_SIZE 1234
IOFUNCATTRDIRTY_TIME 1234, 1386, 1402
iofunc_attr_init() 1228
iofunc_attr_lock() 1230
IOFUNCATTRMTIME 1386, 1402
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOFUNC_ATTRPRIVATE</td>
<td>1234</td>
</tr>
<tr>
<td>IOFUNC_ATTR_T</td>
<td>1335</td>
</tr>
<tr>
<td>Iofunc_attr trylock()</td>
<td>1238</td>
</tr>
<tr>
<td>Iofunc_attr unlock()</td>
<td>1240</td>
</tr>
<tr>
<td>Iofunc_check access()</td>
<td>1243</td>
</tr>
<tr>
<td>Iofunc_chmod()</td>
<td>1246</td>
</tr>
<tr>
<td>Iofunc_chmod_default()</td>
<td>1249</td>
</tr>
<tr>
<td>Iofunc_chown()</td>
<td>1251</td>
</tr>
<tr>
<td>Iofunc_chown_default()</td>
<td>1254</td>
</tr>
<tr>
<td>Iofunc_client_info()</td>
<td>1256</td>
</tr>
<tr>
<td>Iofunc_close dup()</td>
<td>1258</td>
</tr>
<tr>
<td>Iofunc_close dup default()</td>
<td>1261</td>
</tr>
<tr>
<td>Iofunc_close ocb()</td>
<td>1263</td>
</tr>
<tr>
<td>Iofunc_close ocb default()</td>
<td>1265</td>
</tr>
<tr>
<td>Iofunc_devctl()</td>
<td>1267</td>
</tr>
<tr>
<td>Iofunc_devctl default()</td>
<td>1271</td>
</tr>
<tr>
<td>Iofunc_fdinfo()</td>
<td>1274</td>
</tr>
<tr>
<td>Iofunc_fdinfo default()</td>
<td>1276</td>
</tr>
<tr>
<td>Iofunc_func init()</td>
<td>1280</td>
</tr>
<tr>
<td>Iofunc_link()</td>
<td>1284</td>
</tr>
<tr>
<td>Iofunc_lock()</td>
<td>1287</td>
</tr>
<tr>
<td>Iofunc_lock calloc()</td>
<td>1289</td>
</tr>
<tr>
<td>Iofunc_lock default()</td>
<td>1291</td>
</tr>
<tr>
<td>Iofunc_lock free()</td>
<td>1294</td>
</tr>
<tr>
<td>Iofunc_lock ocb default()</td>
<td>1296</td>
</tr>
<tr>
<td>Iofunc_lseek()</td>
<td>1298</td>
</tr>
<tr>
<td>Iofunc_lseek default()</td>
<td>1301</td>
</tr>
<tr>
<td>Iofunc_mkfnod()</td>
<td>1304</td>
</tr>
<tr>
<td>Iofunc_mmap()</td>
<td>1306</td>
</tr>
<tr>
<td>Iofunc_mmap default()</td>
<td>1310</td>
</tr>
<tr>
<td>IOFUNC_MOUNT_32BIT</td>
<td>1299</td>
</tr>
<tr>
<td>IOFUNC_MOUNT_T</td>
<td>1233</td>
</tr>
<tr>
<td>Iofunc_notify()</td>
<td>1314</td>
</tr>
<tr>
<td>IOFUNC_NOTIFY_INPUT</td>
<td>1321</td>
</tr>
<tr>
<td>IFOUNC_NOTIFY OBAND</td>
<td>1321</td>
</tr>
<tr>
<td>IOFUNC_NOTIFY_OUTPUT</td>
<td>1321</td>
</tr>
<tr>
<td>Iofunc_notify_remove()</td>
<td>1319</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iofunc_notify trigger()</td>
<td>1321</td>
</tr>
<tr>
<td>Iofunc_oeb attach()</td>
<td>1325</td>
</tr>
<tr>
<td>Iofunc_oeb calloc()</td>
<td>1326</td>
</tr>
<tr>
<td>Iofunc_oeb detach()</td>
<td>1329</td>
</tr>
<tr>
<td>IOFUNC_OCB FLAGS PRIVATE</td>
<td>1336</td>
</tr>
<tr>
<td>Iofunc_oeb free()</td>
<td>1332</td>
</tr>
<tr>
<td>IOFUNC_OCB PRIVILEGED</td>
<td>1336</td>
</tr>
<tr>
<td>Iofunc_open()</td>
<td>1338</td>
</tr>
<tr>
<td>Iofunc_open default()</td>
<td>1343</td>
</tr>
<tr>
<td>Iofunc opnfd()</td>
<td>1346</td>
</tr>
<tr>
<td>Iofunc opnfd default()</td>
<td>1350</td>
</tr>
<tr>
<td>Iofunc_pathconf()</td>
<td>1352</td>
</tr>
<tr>
<td>Iofunc_pathconf default()</td>
<td>1355</td>
</tr>
<tr>
<td>Iofunc_read default()</td>
<td>1357</td>
</tr>
<tr>
<td>Iofunc_readlink()</td>
<td>1363</td>
</tr>
<tr>
<td>Iofunc_read verify()</td>
<td>1360</td>
</tr>
<tr>
<td>Iofunc_rename()</td>
<td>1367</td>
</tr>
<tr>
<td>Iofunc_space verify()</td>
<td>1371</td>
</tr>
<tr>
<td>Iofunc_stat()</td>
<td>1374</td>
</tr>
<tr>
<td>Iofunc_stat default()</td>
<td>1376</td>
</tr>
<tr>
<td>Iofunc_sync()</td>
<td>1379</td>
</tr>
<tr>
<td>Iofunc sync default()</td>
<td>1381</td>
</tr>
<tr>
<td>Iofunc sync verify()</td>
<td>1383</td>
</tr>
<tr>
<td>Iofunc_time update()</td>
<td>1386</td>
</tr>
<tr>
<td>Iofunc_unblock()</td>
<td>1388</td>
</tr>
<tr>
<td>Iofunc_unblock default()</td>
<td>1390</td>
</tr>
<tr>
<td>Iofunc_unlink()</td>
<td>1394</td>
</tr>
<tr>
<td>Iofunc_unlock ocb default()</td>
<td>1396</td>
</tr>
<tr>
<td>Iofunc utime()</td>
<td>1398</td>
</tr>
<tr>
<td>Iofunc utime default()</td>
<td>1401</td>
</tr>
<tr>
<td>Iofunc write default()</td>
<td>1404</td>
</tr>
<tr>
<td>Iofunc write verify()</td>
<td>1407</td>
</tr>
<tr>
<td>_JOLBF</td>
<td>2962</td>
</tr>
<tr>
<td>_IO_LOCK</td>
<td>1291</td>
</tr>
<tr>
<td>_IO_LSEEK</td>
<td>1298, 1301</td>
</tr>
<tr>
<td>_IO_MMAP</td>
<td>1306, 1310</td>
</tr>
<tr>
<td>_JONBF</td>
<td>2962</td>
</tr>
</tbody>
</table>
anonotify() 1410
_IO_OPENFD 1346, 1350
_IO_OPENFD_NONE 1348
_IO_OPENFD_PIPE 1348
_IO_OPENFD_RESERVED 1348
_IO_PATHCONF 1352, 1355
_IO_READ 1357, 1360
_IO_SET_CONNECT_RET() 1338
_IO_SET_PATHCONF_VALUE() 1352
_IO_SET_READ_NBYTES() 1360
_IO_SET_WRITE_NBYTES() 1407
_IO_SPACE 1371
_IO_STAT 1374, 1376
_IO_SYNC 1381, 1383
_IO_UTIME 1399, 1401
iovec 2508, 2902, 3810
    base, getting 894
    fields, filling 2902
    length, getting 896
    reading from a file 2508
    resource managers 2619
    writing to a file 3810
iovec
    copying 1655
    _IO_WRITE 1404
    _IO_XFLAG_BLOCK 1361, 1408
    _IO_XFLAG_DIR_EXTRA_HINT 1361, 1408
    _IO_XFLAG_NONBLOCK 1361, 1408
    _IO_XTYPE_MQUEUE 1361, 1408
    _IO_XTYPE_NONE 1361, 1408
    _IO_XTYPE_OFFSET 1361, 1408
    _IO_XTYPE_READCOND 1361, 1408
    _IO_XTYPE_REGISTRY 1361, 1408
    _IO_XTYPE_TCPIP 1361, 1408
    _IO_XTYPE_TCPIP_MSG 1361, 1408
    IP (Internet Protocol) 1416
    ip_mreq 1420
    IP addresses
        CIDR (Classless Internet Domain Routing), converting to/from 1105, 1110
classes of 1106
        extracting network number 1108
        local network addresses, extracting 1101
        manipulating 1103
        network numbers, converting to/from strings 1112
        specifying in dot notation 1106
        strings, converting to/from 1097, 1099, 1114, 1116, 1119, 1122
    IP Payload Compression Protocol 1428
    IP6 (Internet Protocol v6) 1442
    IP_ADD_MEMBERSHIP 1420
    IP_DROP_MEMBERSHIP 1420
    IP_HDRINCL 978, 1417
    IP_MAX_MEMBERSHIPS 1420
    IP_MULTICAST_IF 1419
    IP_MULTICAST_LOOP 1419
    IP_MULTICAST_TTL 1418
    IP_OPTIONS 1416
    IPPROTO_ICMPV6 1068, 1069
    IPPROTO_RAW 1417, 1449
    IP_RECV_DSTADDR 1417
    IP_RECVIF 1417
IPsec (secure Internet Protocol) 1423
errors 1440
policies 1431, 1433, 1435
ipsec_dump_policy() 1431
ipsec_get_policylen() 1433
ipsec_set_policy() 1435
ipsec_strerror() 1440
IP_TOS 978, 1416
IP_TTL 1416
IPv6
hop-by-hop and destination options 1131, 1133, 1135, 1137, 1139, 1141
router header options 1149, 1157
ipv6_mreq 1444
IPV6_BINDV6ONLY 1445
IPV6_DSTOPTS 1446
IPV6_HOPLIMIT 1446
IPV6_HOPOPTS 1446
IPV6_JOIN_GROUP 1444
IPV6_MULTICAST_HOPS 1443
IPV6_MULTICAST_IF 1443
IPV6_MULTICAST_LOOP 1443
IPV6_PKTINFO 1445
IPV6_PORTRANGE 1445
IPV6_RTHDR 1446
IPV6_UNICAST_HOPS 1442
IRQs (Interrupt ReQuests), managing 2666
isalnum() 1452
isalpha() 1454
isascii() 1456
isattly() 1458
iscntrl() 1460
isdigit() 1462
isfdtype() 1464
isgraph() 1466
ISIG 3458
isinf(), isnanf() 1468
islower() 1470
isnan(), isnanf() 1472
isprint() 1474
ispunct() 1476
isspace() 1478
ISTRIP 3456
isupper() 1481
iswalnum() 1483
iswalpha() 1485
iswcntrl() 1487
iswctype() 1489
iswdigit() 1491
iswgraph() 1493
iswlower() 1495
iswprint() 1497
iswpunct() 1499
iswspace() 1501
iswupper() 1503
iswxdigit() 1505
isxchar() 1507
ITIMER_REAL 899, 2905
itimerspec 3518
itoa() 1509
IXOFF 3456
IXON 3456

J

j0(), j0f() 1512
j1(), j1f() 1514
jn(), jnf() 1516
job control, supporting 3376
jrand48() 1518
jumps, nonlocal 1578, 2907, 3015, 3061

K

kerinfo 3391
kernel
  blocking states, setting timeouts on 3550
calls
  asyncmsg_connectAttach() 177
  asyncmsg_connectAttr() 179
  asyncmsg_flush() 185
  asyncmsg_connectdetach() 181
  asyncmsg_free() 187
  asyncmsg_get() 189
  asyncmsg_malloc() 191
  ChannelCreate(),
    ChannelCreate_r() 300
  ChannelDestroy(),
    ChannelDestroy_r() 307
  ClockAdjust(),
    ClockAdjust_r() 348
  ClockCycles() 350
  ClockId(), ClockId_r() 353
  ClockPeriod(),
    ClockPeriod_r() 357
  ClockTime(), ClockTime_r() 360
  ConnectAttach(),
    ConnectAttach_r() 382
  ConnectClientInfo(),
    ConnectClientInfo_r() 387
  ConnectDetach(),
    ConnectDetach_r() 303, 390
  ConnectFlags(),
    ConnectFlags_r() 393
  ConnectServerInfo(),
    ConnectServerInfo_r() 395
  DebugBreak() 418
  DebugKDBreak() 420
  DebugKDOoutput() 421
  InterruptAttach(),
    InterruptAttach_r() 1168
  InterruptAttachEvent(),
    InterruptAttachEvent_r() 1176
  InterruptDetach(),
    InterruptDetach_r() 1182
  InterruptHookIdle() 1189
  InterruptHookTrace() 1193
  InterruptMask() 1198
  InterruptUnmask() 1203
  InterruptWait(),
    InterruptWait_r() 1205
  MsgDeliverEvent(),
    MsgDeliverEvent_r() 1788
  MsgError(), MsgError_r() 1796
  MsgInfo(), MsgInfo_r() 1799
Index

MsgKeyData(),
    MsgKeyData_r() 1803
MsgRead(), MsgRead_r()
    1810
MsgReadv(), MsgReadv_r()
    1814
MsgReceive(),
    MsgReceive_r() 1818
MsgReceivePulse(),
    MsgReceivePulse_r() 1824
MsgReceivePulsev(),
    MsgReceivePulsev_r() 1827
MsgReceivev(),
    MsgReceivev_r() 1830
MsgReply(), MsgReply_r()
    1834
MsgReplyv(), MsgReplyv_r()
    1837
MsgSend(), MsgSend_r()
    1842
MsgSendnc(), MsgSendnc_r()
    1847
MsgSendPulse(),
    MsgSendPulse_r() 1851
MsgSendsv(), MsgSendsv_r()
    1855
MsgSendsvnc(),
    MsgSendsvnc_r() 1859
MsgSendv(), MsgSendv_r()
    1863
MsgSendvnc(),
    MsgSendvnc_r() 1867
MsgSendvs(), MsgSendvs_r()
    1871
MsgSendvsnc(),
    MsgSendvsnc_r() 1875
MsgVerifyEvent(),
    MsgVerifyEvent_r() 1878
MsgWrite(), MsgWrite_r()
    1880
MsgWritev(), MsgWritev_r()
    1884
SchedCtl(), SchedCtl_r()
    2746
SchedGet(), SchedGet_r()
    2781
SchedInfo(), SchedInfo_r()
    2785
SchedSet(), SchedSet_r()
    2788
SchedYield(), SchedYield_r()
    2791
SignalAction(),
    SignalAction_r() 3024
SignalKill(), SignalKill_r()
    3032
SignalProcmask(),
    SignalProcmask_r() 3038
SignalSuspend(),
    SignalSuspend_r() 3042
SignalWaitinfo(),
    SignalWaitinfo_r() 3045
SyncCondvarSignal(),
    SyncCondvarSignal_r() 3341
SyncCondvarWait(),
    SyncCondvarWait_r() 3344
SyncCtl(), SyncCtl_r() 3350
SyncDestroy(),
    SyncDestroy_r() 3352
SyncMutexEvent(),
SyncMutexEvent_r()  3355
SyncMutexLock(),
SyncMutexLock_r()  3358
SyncMutexRevive(),
SyncMutexRevive_r()  3361
SyncMutexUnlock(),
SyncMutexUnlock_r()  3363
SyncSemPost(),
SyncSemPost_r()  3366
SyncSemWait(),
SyncSemWait_r()  3368
SyncTypeCreate(),
SyncTypeCreate_r()  3372
ThreadCancel(),
ThreadCancel_r()  3479
ThreadCreate(),
ThreadCreate_r()  3484
ThreadCtl(), ThreadCtl_r()  
1168, 1177, 1182, 1185,
1187, 1195, 1198, 1201,
1203, 3490
ThreadDestroy(),
ThreadDestroy_r()  3498
ThreadDetach(),
ThreadDetach_r()  3501
ThreadJoin(), ThreadJoin_r()  3503
TimerAlarm()  3532
TimerCreate()  3535
TimerDestroy(),
TimerDestroy_r()  3538
TimerInfo(), TimerInfo_r()  3542
TimerSettime(),
TimerSettime_r()  3546
TimerTimeout(),
TimerTimeout_r()  3550
TraceEvent()  3582
debugging  420, 421
instrumented  1193, 3582
thread scheduler  2785
kill()  1520
killpg()  1523

L

labs()  1525
large-file support  104
file information  782, 1590,
3230
filesystem information  786,
3238
mapped memory, offset
of  1640
opening  404, 1974
position
setting  1587, 3448
reading  2123
shared memory, mapping  1704
symbolic link
information  1590
system-resource limits  956,
2938
truncating  798
writing  2445
LC_ALL  2914
LC_COLLATE  2914
LC_CTYPE  1490, 2914
Index

host names, length 2900
hostnames, length 2900
inheritance 587
iov arrays 2508
memory locked into physical memory 2938
path names, length of 746, 2009
pipes, number of bytes written atomically 746, 2009
processes
  argument lists 3375
  CPU time 2938
  execution time 2735
  files, number open 3376
  I/O requests 1547
  mapped address space 2938
  maximum per real user ID 2938, 3375
  scheduling policy 2723, 2725
  supplementary group IDs 3375
resident set size 2938
sockets, pending connections 1552, 2648
stack size 2938
system resources
  getting 845, 956
  setting 2938
TCP maximum segment size 3433
threads

Index November 2, 2006
execu‌tion time 2785
maximum number of 2938
priority 2785
stack size 2202
_LINE__ 168
line buffering, setting for stream
I/O 2912
linear search 1536, 1583
LINES 326
link() 1543
resource managers,
implementing in 1284
link-local addresses 1128
linker symbols
.btext 253
.cdata 519
.end 522
.etext 583
LINK_MAX 704, 1544
links, symbolic
creating 3337
deleting 2542, 3631
information, getting 1590
ownership, changing 1527
reading 2505
resolving 2515
temporary 2016, 2018
lio_listio() 1548
LIO_NOP 1548
LIO_NOWAIT 1547
LIO_READ 1548
LIO_WAIT 1547
LIO_WRITE 1548
listen() 1552, 3432, 3825
little endian
big endian, converting
to/from 539, 541, 543,
545, 547, 549, 3331
__LITTLEENDIAN__ manifest
113
messages 1666
native format, converting
to/from 533, 535, 537
ports
reading from 1087, 1091
writing to 1996, 2000
unaligned values
accessing safely 3615, 3617,
3619
writing safely 3609, 3611,
3613
__LITTLEENDIAN__ 113
lltoa() 1594
local network addresses, converting
to/from IP addresses
1103
local times, converting to/from
calendar times 1559,
1561, 1693
LOCAL_CREDs 3630
LOCALDOMAIN 2548, 2552,
2556, 2558, 2561, 2564
localeconv() 1554
locales
classes, wide-character 3784
daylight saving time 417
numeric formatting 1554
setting 2915
strings, comparing 3258, 3733
localtime() 1559
localtime_r() 1561
LOCK_EX 706

November 2, 2006
lockf() 1564
LOCK_NB 706
locks
files 661, 706, 709, 801, 1564
mutexes
attributes, destroying 2304
attributes, initializing 2318
attributes, priority ceiling 2306, 2321
attributes,
process-shared 2311, 2326
attributes, recursive 2313, 2327
attributes, scheduling protocol 2308, 2324
attributes, type 2315, 2330
destroying 2285, 3352
events 3350, 3355
initializing 2289, 3372
locking 2291, 2297, 2300, 3358, 3363
priority 3350
priority ceiling 2287, 2295
reviving 3361
unlocking 2302
read-write attributes, creating 2360
attributes, destroying 2356
attributes,
process-shared 2358, 2362
destroying 2335, 3352
initializing 2337, 3372
locking for reading 2340, 2342, 2348
locking for writing 2345, 2350, 2354
unlocking 2352
sleepon
destroying 3085
initializing 3087
locking 2381, 3089
unblocking 2379, 2383, 3083, 3091
unlocking 2389, 3093
waiting 2385, 2391, 3095
LOCK_SH 706
LOCK_UN 706
log
bankruptcy (not implemented) 2752
log(), logf() 1567
LOG_*(logging facilities) 1988
log, system message
closing 368
log priority mask 2917
opening 1989
writing to 3386, 3699
blocks 3097
formatted output 3100, 3685
integers 3103
log10(), log10f() 1571
log1p(), log1pf() 1569
LOG_ALERT 3387
logarithms
base 10 1571
integral 1081
natural 1567
x + 1 1569
logb(), logbf() 1573
LOG_CRIT 3387
LOG_DEBUG 3387
LOG_EMERG 3387
LOG_ERR 3387
logging in
   previous lines, discarding 1722
   pseudo-ttys 731, 1576
logical interrupt vector
   numbers 1168
LOG_INFO 3387
LOGIN_PROCESS 1005, 1008, 3644
login_tty() 1576
LOG_MASK() 2917
LOG_NOTICE 3387
LOG_UPTO() 2917
LOG_WARNING 3387
long integers
   absolute value 1525
division 1534
pseudo-random numbers
   nonnegative 1581, 1963
   signed 1518, 1783, 2464
   quotient 1534
   remainder 1534
   size of 113
_LONG_BITS_ 113
longjmp() 1578, 2457
lowercase
   characters, converting to 3572
   strings, converting to 3278
testing a character for 1470, 1495
   wide characters, converting to 3576, 3578, 3782
lrand48() 1581
lsearch() 1583
lseek(), lseek64() 1587
resource managers,
   implementing in 1298, 1301
lstat(), lstat64() 1590
L_tmpnam 3569
ltoa() 1594
ltrunc() 1597

M
Machine Status Register 3391
main() 1604, 3019
   arguments
      auxiliary 234
      number of 159
      parsing 916
      vector of 160
major device numbers 2591, 2681, 3234
major() 3234
_MAJOR_BLK_PREFIX 2682
_MAJOR_CHAR_PREFIX 2682
_MAJOR_DEV 2682
_MAJOR_FSYS 2682
_MAJOR_PATHMGR 2682
makedev() 3234
mallinfo 1607
mallinfo() 1607
malloc() 1609
mallopt() 1612
manifests 113
MAP_ANON 1705
MAP_BELOW16M 1705
MAP_FAILED 1707, 1714
Index

© 2006, QNX Software Systems GmbH & Co. KG.

MAP_FIXED 1704, 1705, 1713, 1714
MAP.LAZY 1705
MAP_NOX64K 1706
MAP_PHYS 1706
MAP_PRIVATE 1704
MAP_SHARED 1705
MAP_STACK 1706
MAP_TYPE 1704
mathematics
   absolute values 121, 258, 646, 1525
   Bessel functions 1512, 1514, 1516, 3815, 3817, 3819
   complementary error function 569
   division 468, 1534
   error function 567
   exponentials 638, 640, 773, 1532, 2699, 2702, 3048
   finite numbers 701
   floating-point settings 734, 736, 739, 742
   gamma functions 822, 1539
   hyperbolic functions 131, 166, 200, 402, 3079, 3406
   hypotenuse, length of 1060
   infinite numbers 1468
   logarithms 1081, 1567, 1569, 1571
   maximum 1616
   minimum 1677
   modular arithmetic 715
   next representable number 1954
   not a number, determining if 1472
   powers 2121
   pseudo-random numbers 491, 565, 1161, 1518, 1530, 1581, 1783, 1963, 2460, 2462, 2464, 2820, 2953, 3219, 3221, 3223
   radix-independent exponents 1573
   remainders 493, 2540
   roots 276, 3217
   rounding 278, 711, 2645
   sign, copying 398
   trigonometry 129, 164, 196, 198, 400, 3077, 3404
max() 1616
MAXHOSTNAMELEN 2900
Maximum Segment Size (MSS) 3433
MB_CUR_MAX 3779
mblen() 1618
mbrlen() 1621
mbrtowc() 1623
mbsinit() 1626
mbsrtowcs() 1628
mbstate_t 1626
mbstowcs() 1630
mbtowc() 1633
mcheck_status 1746
mcheck() 1636
MCHECK_DISABLED 1746
MCHECK_FREE 1746
MCHECK_HEAD 1746
MCHECK_OK 1746
MCHECK_TAIL 1746
memalign() 1643
members, offset of within a structure 1971
memccpy() 1645
memchr() 1647
memcmp() 1650
memcpy() 1653
memcpyv() 1655
memicmp() 1657
memmove() 1660
mem_offset(), mem_offset64() 1640
memory
allocating
aligned 1643, 2111
__amblksize 3105
array 274, 2705
automatic (from stack) 152
blocks 1609, 2512, 3105
break value, changing 157, 2696
consistency check 1636, 1745
controlling 1612
data segment, changing 247
heap block, aligned on page boundary 3663
information about,
getting 1607
comparing 1650, 1657, 3788
copying 1645, 1653, 1655, 3790
overlapping objects 1660, 3792
devices
I/O, mapping 1710, 1897
physical, mapping into process’s address space 1713
direct memory access (DMA)
channels, managing 2666
free, amount of 3236
freeing 292, 763, 2512, 2965, 3225
locking 1699, 1891
maximum amount 2938
managing 2666
mapped
contiguous length 1640
maximum size 2938
offset of 1640
physical storage, synchronizing with 1888
reallocating 2512, 3225
searching
for a character 1647
for a wide character 3786
setting 1662, 3794
shared
mapping 1704
unmapping 1895
unlocking 1893
unmapping 1899
memset() 1662
message_attr_t 1665
message queues
attributes 1753, 1771
closing 1751
messages
receiving 1765, 1775
sending 1768, 1778
notifying when
nonempty 1757
opening 1762
persistence of 1763
receive-only 1760
send-only 1760
send-receive 1760

November 2, 2006
unlinking 1780
message_attach() 1665
message_connect() 1671
message_detach() 1674
messages
    channels
        attaching to a process 382
        creating 300, 1904
        destroying 307, 1913
    dispatch interface
        handlers 1665, 1674
    errors, handling 1796
    information about
        getting 1799
        structure 1785
Internet domain name servers
    errors 1030, 1036, 1045
    queries 2552, 2555, 2558,
    2561, 2563
    sending and
        interpreting 487, 489,
        2548
key, adding 1803
reading data 1810, 1814, 2611,
    2613
receiving 1818, 1830
replying 1834, 1837
resource managers
    blocking while waiting
        for 2575
    handling 2599
sending 1842, 1847, 1855,
    1859, 1863, 1867, 1871,
    1875
SNMP
    creating 3118
    freeing 3109
    reading 3120
    sending 3126
    sockets
        peeking at 2517, 2520, 2524
        receiving from 2518, 2521,
        2525
        sending to 2866, 2869, 2872
        tampering, preventing 1803
        unblocking 1796
        writing data 1880, 1884, 2615,
        2617
    MFLAG_OCB 1739, 1742
    min() 1677
    minor device numbers 2591, 2681,
    3234
    minor() 3234
    misaligned access response 3490
mkdir() 1679, 3605
mkfifo() 1682, 3605
mknod() 1234, 1685
    resource managers,
        implementing in 1304
mkstemp() 1689
mktemp() 1691
mktime() 1693
mlockall() 1699
mmap(), mmap64() 1704
    resource managers,
        implementing in 1306,
        1310
mmap_device_iot() 1710
mmap_device_memory() 1713
modem_script 1726
MODEM_ALLOW8BIT 1721
MODEM_ALLOWCASE 1721
MODEM_ALLOWCTRL 1721
MODEM_BAUD 1727, 1731
MODEM  LASTLINE  1722
MODEM_NOECHO  1727, 1728, 1731
modem_open()  1716
modem_read()  1722
modems
  opening  1716
  reading  1722
  script, running on  1726
  states  1728
  writing  1733
  escape characters  1733
  special characters  1733
modem_script()  1726
modem_write()  1733
modify(), modifie()  1736
modular arithmetic, floating point  715
mount()  1740
  _MOUNT_AFTER  1740, 1742
  _MOUNT_ATIME  1742
  _MOUNT_BEFORE  1739, 1742
  _MOUNT_CREAT  1742
  _MOUNT_ENUMERATE  1740, 1742
  _MOUNT_FORCE  1740, 1742, 3607
  _MOUNT_NOATIME  1739, 1742
  _MOUNT_NOCREAT  1739, 1742
  _MOUNT_NOEXEC  1268, 1739, 1742
  _MOUNT_NOSUID  1268, 1739, 1742
  _MOUNT_OFF32  1739, 1742
  _MOUNT_OPAQUE  1740, 1742
mount_parse_generic_args()  1742
  _MOUNT_READONLY  1268, 1739, 1742
  _MOUNT_REMOUNT  1740, 1742
  _MOUNT_SUID  1742
  _MOUNT_UNMOUNT  1740, 1742
mprobe()  1745
mprotect()  1748
mq_attr  1753
  mq_close()  1751
  mq_getattr()  1753
  mq_notify()  1757
  mq_open()  1762
MQ_PRIOR_MAX  1768
mq_receive()  1765
mq_send()  1768
mq_setattr()  1771
mq_timedreceive()  1775
mq_timedsend()  1778
mq_unlink()  1780
mrand48()  1783
MS_ASYNC  1887
MSG_ABORT  2814
MSG_ADDR_OVER  2814
MSG_CTRUNC  2526
MsgDeliverEvent(),
  MsgDeliverEvent_r()  1412, 1788
MSG_DONTROUTE  2865, 2868, 2871
MSG_EOF  2814
MSG_EOR  2526
MsgError(), MsgError_r()  1796
MSG_FLAG_CROSS_ENDIAN  460
MSG_FLAG_SIDE_CHANNEL  1671
msghdr  2525
MsgInfo(), MsgInfo_r()  1799
MsgKeyData(), MsgKeyData_r()  1803
MSG_OOB  981, 2517, 2520, 2524, 2526, 2865, 2868, 2871

November 2, 2006
MSG_PEEK  2517, 2520, 2524
MsgRead(), MsgRead_r()  1810
MsgReadv(), MsgReadv_r()  1814
MsgReceive(), MsgReceive_r()  1818
MsgReceivePulse(),
    MsgReceivePulse_r()  1824
MsgReceivePulsev(),
    MsgReceivePulsev_r()  1827
MsgReceivev(), MsgReceivev_r()  1830
MsgReply(), MsgReply_r()  1834
MsgReplyv(), MsgReplyv_r()  1837
MsgSend(), MsgSend_r()  1842
MsgSendnc(), MsgSendnc_r()  1847
MsgSendPulse(), MsgSendPulse_r()  1851
MsgSendsv(), MsgSendsv_r()  1855
MsgSendsvnc(), MsgSendsvnc_r()  1859
MsgSendv(), MsgSendv_r()  1863
MsgSendvnc(), MsgSendvnc_r()  1867
MsgSendvs(), MsgSendvs_r()  1871
MsgSendvsnc(), MsgSendvsnc_r()  1875
MSG_TRUNC  2526
MSG_UNORDERED  2813
MsgVerifyEvent(),
    MsgVerifyEvent_r()  1878
MSG_WAITALL  2517, 2521, 2524
MsgWrite(), MsgWrite_r()  1880
MsgWritev(), MsgWritev_r()  1884
MS_INVALIDATE  1887
MS_INVALIDATE_ICACHE  1887

MSS (Maximum Segment
    Size)  3433
MS_SYNC  1887
msync()  1888
multibyte characters
    bytes, counting  1618, 1621
    wide characters
        conversion object  1626
        wide characters, converting
to/from  1623, 1628, 1630,  
    1633, 3725, 3753, 3769,  
    3779
munlock()  1891
munlockall()  1893
munmap()  1895
munmap_device_io()  1897
munmap_device_memory()  1899
mutexes
    attributes
        destroying  2304
        initializing  2318
        priority ceiling  2306, 2321
        process-shared  2311, 2326
        recursive  2313, 2327
        scheduling protocol  2308,  
        2324
        type  2315, 2330
        destroying  2285, 3352
events, attaching  3350, 3355
        initializing  2289, 3372
locking  2291, 2297, 2300, 
        3358, 3363
    priority  3350
    ceiling  2287, 2295
    reviving  3361
    unlocking  2302
N

name_attach_t 1905
name servers
  errors 1030, 1036, 1045
  initializing 2548
  names
    compressing 487
    expanding 489
  queries 2552, 2555, 2558, 2561, 2563
name_attach() 1904
NAME_ATTACHMENT_FLAG_GLOBAL 1905
name_close() 1911
name_detach() 1913
NAME_FLAG_ATTACHMENT_GLOBAL 1904, 1915
NAME_FLAG_DETACH_SAVEDPP 1913
NAME_MAX 1916
name_open() 1915
names
  binding to sockets 242, 2467
  domain
    setting 2880
  domain, getting 374, 843
  host
    getting 890
    setting 2900
  peer, getting 924
  socket, getting 974, 2641
NAN (not-a-number) 1472
nanoseconds
  busy-waiting for 1920, 1923, 1926, 1928, 1930
  threads, suspending for 1918
timespec, converting to/from 1965, 3561
nanosleep() 1918
nanospin() 1920
nanospin_calibrate() 1923
nanospin_count() 1926
nanospin_ns() 1928
nanospin_ns_to_count() 1930
nap() 1933
napms() 1935
natural logarithms 1567
nbacconnect() 1937
nbacconnect_result() 1940
ND2S_DIR_HIDE 1946
ND2S_DIR_SHOW 1946
ND2S_DOMAIN_HIDE 1946
ND2S_DOMAIN_SHOW 1947
ND2S_LOCAL_STR 1947
ND2S_NAME_HIDE 1947
ND2S_NAME_SHOW 1947
ND2S_QOS_HIDE 1947
ND2S_QOS_SHOW 1947
ND2S_SEP_FORCE 1947
NDEBUG 168
ND_LOCAL_NODE 1942
ND_NODE_CMP() 1942
NETDB_INTERNAL 880, 1030, 1037
netent 1944
netmgr.ndtostr() 1945
netmgr_remote_nd() 1950
netmgr_strnd() 1952
network
database
  closing 551
  entries, getting 910, 912, 914
  opening 2919
  structure 1944
Index

host entries
  errors 1030, 1036, 1045
  getting 873, 877, 879, 883, 885, 888
network interface addresses
  freeing 767
  getting 892
  structure 1079
network numbers
  IP addresses, converting
to/from 1103, 1108
  strings, converting
to/from 1112
network-byte order
  host-byte order, converting
to/from 1047, 1049, 1967, 1969
Neutrino classification 107
NEW_TIME 1005, 3644
nextafter(), nextafterf() 1954
nftw(), nftw64() 1957
NGROUPS_MAX 386, 871, 2896
nice() 1961
NO_DATA 1031, 1037
node descriptors
  comparing 1942
  current 3621
  relative to a remote node 1950
  strings, converting
to/from 1945, 1952
nodenames, translating addresses
  into 905
NOFD 1640, 1704, 1705
NOFLSH 3458
nonlocal jumps 1578, 2907, 3015, 3061
NO_RECOVERY 1031, 1037
normalized fractions 773
not a number, determining if 1472
_NOTIFY_ACTION_POLL 1413
_NOTIFY_ACTION_POLLARM 1413
_NOTIFY_ACTION_TRANARM 1413
_NOTIFY_COND_INPUT 1315, 1411
_NOTIFY_COND_MASK 1412
_NOTIFY_COND_OBAND 1316, 1411
_NOTIFY_COND_OUTPUT 1316, 1411
_NOTIFY_DATA_MASK 1412
nrand48() 1963
nsec2timespec() 1965
NSIG 3006
_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, 1786
_NTO_CHF_SENDER_LEN 304, 1786
_NTO_CHF_THREAD_DEATH 304
_NTO_CHF Unblock 304, 305, 1786
_NTO_COF_CLOEXEC 382, 392
ntohl() 1967
ntohs() 1969
_NTO_INTR_CLASS_EXTERNAL 1168
_NTO_INTR_CLASS_SYNTHETIC 1168
_NTO_INTR_FLAGS_END 1172, 1173, 1178, 1190
_NTO_INTR_FLAGS_PROCESS 1172, 1173, 1178, 1190, 1191
Index

_NTO_INTR_FLAGS_TRK_MSK 1172, 1190
_NTO_INTR_SPARSE 1168
_NTO_KEYDATA_CALCULATE 1802, 1805
_NTO_KEYDATA_VERIFY 1802, 1805
_NTO_MI_UNBLOCK_REQ 1786
_NTO_RESET_OVERRUNS 3541
_NTO_SCTL_GETPRIORITYCEILING 3349
_NTO_SCTL_SETEVENT 3349
_NTO_SCTL_SETPRIORITYCEILING 3349
_NTO_SYNC_COND 3371
_NTO_SYNC_MUTEX_FREE 3371
_NTO_SYNCSEM 3371
_NTO_TCTL_ALIGN_FAULT 3490
_NTO_TCTL_IO 1168, 1177, 1182, 1185, 1187, 1195, 1198, 1201, 1203, 3492
_NTO_TCTL_RUNMASK 3492
_NTO_TCTL_THREADS_CONT 3494
_NTO_TCTL_THREADS_HOLD 3494
_NTO_TIMEOUT_CONDVAR 3551
_NTO_TIMEOUT_INTR 3551
_NTO_TIMEOUT_JOIN 3551
_NTO_TIMEOUT_MUTEX 3551
_NTO_TIMEOUT_RECEIVE 3551
_NTO_TIMEOUT_REPLY 3551
_NTO_TIMEOUT_SEM 3551
_NTO_TIMEOUT_SEND 3551
_NTO_TIMEOUT_SIGSUSPEND 3552
_NTO_TIMEOUT_SIGWAITINFO 3552
_NTO_TIMER_SEARCH 3541
_NTO_TRACE.* 3584
_NTO_VERSION 113
_numbers
determining if

finite 701
infinite 1468
not a number 1472
formatting 1554
maximum 1616
minimum 1677
next representable
floating-point 1954
strings, converting
to/from 205, 207, 209, 211, 1509, 1594, 3310, 3314, 3321, 3324, 3601, 3648
wide-character strings,
converting to/from 3759, 3763, 3767, 3773

O

O_APPEND 661, 1214, 1348, 1974, 1984, 3149, 3154, 3802
OCBs (Open Control Blocks)
allocating 1326
attaching 1325, 2623
detaching 1263, 1265, 1329, 2633
freeing 1265, 1332
getting 2621
structure 1335
unlocking 1396
O_CLOEXEC 1974
O_CREAT 404, 1760, 1974, 2979, 3149, 3150
O_DSYNC 1214, 1379, 1384, 1975, 3808
optimization, compiling with 113
    __OPTIMIZE__ 113
optind 916
options
    command-line
        parsing 916, 994
    mount, parsing 1742
    socket-level 977, 2950
optopt 917
O_RDONLY 661, 1347, 1760, 1974,
        1984, 2979, 3149, 3154
O_RDWR 661, 1348, 1597, 1760,
        1974, 1984, 2033, 2979, 3149, 3154
O_REALIDS 1215, 1256, 1976
O_SYNC 1215, 1379, 1384, 1977,
        3808
other scheduling 2192
O_TRUNC 404, 1215, 1348, 1977,
        1984, 2979, 3149, 3154
out-of-band (OOB) data
    determining if at mark 3138
    sending/receiving 2517, 2520,
        2524
out16() 1996
out16s() 1998
out32() 2000
out32s() 2002
out8() 1992
out8s() 1994
outbe16() 1996
outbe32() 2000
outle16() 1996
outle32() 2000

0_EXCL 1215, 1760, 1975, 2033,
        2979, 3149
off_t, limiting to 32 bits 1739
offsetof() 1971
OHFLOW 3457
O_LARGEFILE 1215, 1975
OLD_TIME 1005, 3644
once-initialization 2332
O_NOCTTY 1215, 1975
O_NONBLOCK 661, 1359, 1370,
        1406, 1723, 1760, 1975,
        2094, 2481, 2496, 2510,
        3803
OOB (out-of-band) data
    determining if at mark 3138
    sending/receiving 2517, 2520,
        2524
Open Control Blocks  See OCBs
open(), open64() 404, 1974, 3605
    resource managers,
    implementing in 1343
opendir() 1981
openfd() 1984
    resource managers,
    implementing in 1346,
        1350
openlog() 1989
OPEN_MAX 2942, 3157, 3188
openpty() 1991
operating system
    name 374, 3621
    release level 374
    target 113
    version 113, 374, 3621
OPOST 3457
optarg 916
opterr 917
output, formatted 748, 813, 2126, 3669, 3675, 3679, 3703, 3800
overlapping memory, copying 1660, 3792
ownership, changing of a file 317, 653, 1527
O_RDONLY 404, 661, 1347, 1597, 1760, 1974, 1984, 3149, 3154

P

packets See also ROUTE
  routing 2654
SNMP
    reading 3120
P_ALL 3717
PARENB 3457
PARMRK 3456
PARODD 3457
PARSTK 3457
partitions, adaptive See adaptive partitions
passwd 942, 945, 948, 950, 953
passwords
  database
    closing 554
    entries, getting for a user 945, 948, 950, 953
    entries, getting next 942
    rewinding 2930
    encrypting 408, 2448
    prompting for and reading 922
shadow database
  closing 557
  entry, reading 690, 987, 992
  entry, structure 2432
  entry, writing 2432
  rewinding 2952
PATH 57, 326, 605, 628, 2817, 3178, 3183, 3188, 3205, 3209
pathconf( ) 2008
  resource managers,
    implementing in 1352, 1355
pathfind( ), pathfind_r( ) 2013
PATH_MAX 840, 1916
pathmgr_symlink( ) 2016
pathmgrunlink( ) 2018
pathname delimiter in QNX
  Momentics documentation
live
paths
  base name 235
  directory name 440
  names
    maximum length 746, 2009
    patterns, matching 719, 1020, 1024
    truncating 746, 2009
    resolving 2515
resource managers
  attaching to 2568
  detaching from 2587
  getting 2626
_PATH_UTMP 559
pattern matching See regular expressions
pause( ) 2020
PC Card server
Index

© 2006, QNX Software Systems GmbH & Co. KG.

arming 2023
attaching 2026
card insertion/removal,
notification of 2023
CIS (Card Information
Structure), reading 2036
detaching 2028
locking 2033
socket setup information 2030
unlocking 2037

pccard_arm() 2023
_PCCARD_ARM_INSERT_REMOVE 2022
pccard_attach() 2026
pccard_detach() 2028
_PCCARD_DEV_AIMS 2022
_PCCARD_DEV_ALL 2022
_PCCARD_DEV_FIXED_DISK 2022
_PCCARD_DEV_GPIB 2022
_PCCARD_DEV_MEMORY 2022
_PCCARD_DEV_NETWORK 2022
_PCCARD_DEV_PARALLEL 2022
_PCCARD_DEV_SCSI 2022
_PCCARD_DEV_SERIAL 2022
_PCCARD_DEV_SOUND 2022
_PCCARD_DEV_VIDEO 2022
pccard_info() 2030
pccard_lock() 2033
_PCCARD_MEMTYPE_ATTRIBUTE 2035
_PCCARD_MEMTYPE_COMMON 2035
pccard_raw_read() 2036
pccard_unlock() 2037
_PC_CHOWN_RESTRICTED 746, 2009

PCI
addresses
converting 2046
testing 2046

BIOS, determining if
present 2066
classes, finding 2055
devices
attaching 2041
configuration, reading 2068,
2071, 2074, 2077
configuration, writing 2081,
2084, 2086, 2088
detaching 2053
finding 2055, 2058
rescanning for 2079
functions, finding 2055
interrupts
mapping 2063
routing information 2060
memory, sharing 1707
server
attaching 2039
detaching 2051

pci_dev_info 2042
pci_attach() 2039
pci_attach_device() 2041
PCI_BAD_REGISTER_NUMBER 2069,
2072, 2075, 2078, 2085,
2087, 2089
PCI_BUFFER_TOO_SMALL 2069,
2072, 2075, 2085, 2087
pci_detach_device() 2053
PCI_DEVICE_NOT_FOUND 2053,
2056, 2058, 2069
pci_find_class() 2055
pci_find_device() 2058
PCI_INIT_ALL 2046
PCI_INIT_BASE0 ... 
PCI_INIT_BASE5 2046
PCI_INIT_IRQ 2046

3948 Index

November 2, 2006
PCI_INIT_ROM 2046
PCI_IO_ADDR() 2046
pci_irq_routing_options() 2060
PCI_IS_IO() 2046
PCI_IS_MEM() 2046
pci_map_irq() 2063
PCI_MASTER_ENABLE 2046
PCI_MEM_ADDR() 2047
PCI_PERSIST 2045, 2053
pci_present() 2066
pci_read_config() 2068
pci_read_config16() 2074
pci_read_config32() 2077
pci_read_config8() 2071
pci_rescan_bus() 2079
PCI_ROM_ADDR() 2047
PCI_SEARCH_BUSDEV 2046
PCI_SEARCH_CLASS 2045
PCI_SEARCH_VEND 2045
PCI_SEARCH_VENDEV 2045
PCI_SET_FAILED 2064
PCI_SHARE 2045
PCI_SUCCESS 2056, 2058, 2066, 2072, 2075, 2078, 2085, 2087, 2089
PCI_UNSUPPORTED_FUNCTION 2064
pci_write_config() 2081
pci_write_config16() 2086
pci_write_config32() 2088
pci_write_config8() 2084
_PC_LINK_MAX 745, 2008
pclose() 2090, 2105
_PC_MAX_CANON 745, 2008
_PC_MAX_INPUT 745, 2008
_PC_NAME_MAX 745, 2009
_PC_NO_TRUNC 746, 2009
_PC_PATH_MAX 746, 2009
_PC_PIPE_BUF 746, 2009
_PC_VDISABLE 746, 2009, 3459
PDU (Protocol Data Unit) See SNMP
peers, getting names of connected 924
Peripheral Component Interconnect See PCI
permissions
changing 313, 650
files, on creation 3604
daemons 2136
perror() 2092
PF_INET 3381
PF_KEY 1423
PF_KEY_V2 1423
PF_ROUTE 1128
pipe() 2094
PIPE_BUF 3803
pipes
bytes, writing atomically 746, 2009
closing 2090
creating 2094
opening 2104
reading from 2123, 2480, 2508
P_NOWAIT 3168, 3173, 3178, 3183, 3195, 3200, 3205, 3209
P_NOWAITO 3168, 3173, 3178, 3183, 3195, 3200, 3205, 3209
pointers, size of void 113
poll() 2097
pollfd 2097
POOL_FLAG_EXIT_SELF 3463
POOL_FLAG_USE_SELF 3463
Index

`popen()` 2104
portable code 104
ports
  managing 2666
  privileged
    socket, binding to 245
    socket, getting for 2662
  reading from 1083, 1085, 1087, 1089, 1091, 1093
serial
  opening 1716
  reading 1722
  script, running on 1726
  writing 1733
services, finding for 968
POSIX See also message queues; semaphores; threads
  signals 3024
  standards 104
  version supported 3376
  `_POSIX_AIO_MAX` 1547
  `_POSIX_CHOWN_RESTRICTED` 318, 653, 1528
  `_POSIX_LOGIN_NAME_MAX` 903
  `posix_memalign()` 2111
  `_POSIX_THREAD_SAFE_FUNCTIONS` 860, 903
  `P_OVERLAY` 49, 3168, 3173, 3178, 3183, 3195, 3200, 3205, 3209
  `pow()`, `powf()` 2121
PowerPC platforms, variable-length argument lists on 3652
powers 2121
  `P_PGID` 3717
  `P_PID` 3717
  `pread()`, `pread64()` 2123
precision, floating-point 739
printable, testing a character for 1466, 1474, 1493, 1497
`printf()` 2126
priorities
  adjusting 1961, 2721
  getting 934, 2718
  maximum 2723
  minimum 2725
  setting 2737, 2741, 2926, 3162
process groups
  changing 2150
  creating 2921, 2946, 3163
  devices 3421, 3441
  ID, getting 926, 928
  joining 2921
  membership, inheriting 586, 593, 601, 611, 617, 623, 3159
  pulses, sending 1851
  remote node 3161
  session of a controlling terminal 3423
  setting 2924, 3162
  signals, sending 1520, 1523, 2993, 3029, 3032
  `SIGHUP` 633
  `SIGURG` 3142
  status of 3713, 3720
  waiting for 3717
processes See also threads
  adaptive partition
    getting 2777
    setting 2759
address space
  device I/O memory,  mapping  1713
  limits  2938
  locking  1699
  unlocking  1893
alarms, scheduling  149, 3596
alignment  3161
analyzing  1193, 3582
arguments
  auxiliary  234
  maximum length  3375
  number of  159
  parsing  916
  vector of  160
asynchronous message channels,
  attaching to  177
background  415, 2136
  termination, notification  2138
buffers
  allocating  191
child
  closing file descriptors  in  3157, 3188
  state change, waiting  for  3707, 3711, 3714,
  3718, 3721
  zombie, preventing from becoming  2992, 3027,
  3162
configurable limits  3375
connections
  client, information  about  387
  detaching  181, 303, 390
  flags, modifying  393
flush  185
information about  395
receiving  189
controlling terminal, path name  410
CPU time, maximum  2938
creating  584, 592, 599, 605,
  609, 615, 621, 628, 727,
  731, 3158, 3162, 3169,
  3174, 3179, 3184, 3189,
  3196, 3201, 3206, 3210
data segment, changing space allocated for  247
debugging  418, 3162
dynamically linked libraries
  addresses, translating  471
  closing  473
  debugging  482
  errors  475
  opening  477
  symbol, getting address of  484
environment  561
  clearing  326
  restoring  1578, 3015
  saving  2907, 3061
environment variables
  defining  2427, 2885
  deleting  2427, 2885, 3634
  getting  849
executable file
  base name  2152
  file descriptor  369
  full path  370
executing  584, 592, 599, 605,
  609, 615, 621, 628, 3162
execution time  353
Index

© 2006, QNX Software Systems GmbH & Co. KG.

execution time limit,
  getting 2735
file-mode creation mask 3604
daemons 2136
files, maximum per 3376
forking 727, 731
group ID
effective 847, 2882, 2891, 2931
real 853, 2891, 2931
saved 2891
supplementary 871, 2896
guardian, specifying 2145
holding for debugging 3162
I/O privileges,
  requesting 1168, 1177, 1182, 1185, 1187, 1195, 1198, 1201, 1203, 3492
ID, getting 930, 932
interrupts
  disabling 1185
  enabling 1187
  events 1176, 1182
  handlers 1168, 1182, 1195, 1201
  handlers, idle 1189
  masking 1198, 1203
  waiting for 1205
maximum per real user
  ID 2938, 3375
memory, sharing 1706
message channels, attaching to 382
name 370, 2152
parent
  blocking 3667
  ID, getting 932
priority
  adjusting 1961, 2721
  getting 934, 2718
  maximum 2723
  minimum 2725
  setting 2737, 2741, 2926, 3162
processor affinity 3492
program entry function 1604
resident set size,
  maximum 2938
scheduling parameters 3162
scheduling policy
  getting 2727
  setting 2741, 3162
sessions 972, 2924, 2946
set-group ID 3376
set-user ID 3376
SIGALRM sending to 3509, 3535, 3596
signals
  actions for 2990, 3019, 3024
  information about 3075
  pending 3053
  queuing 3058
  raising 2457
  sending 1520, 1523, 3032
  suspending until delivered 3042
  waiting for 2020, 3045, 3068, 3073, 3075
spawning 3158, 3169, 3174, 3179, 3184, 3189, 3196, 3201, 3206, 3210
spawning and blocking 3667
stack size, setting maximum for 3163
supplementary group IDs, maximum 3375
suspending 423, 1918, 1933, 1935, 3081, 3636, 3707, 3711, 3714, 3718, 3721
system commands, executing 3397
system-wide events
notification of 2139
triggering 2143
terminating 119, 632, 635
diagnostics 168
functions to be called, registering 202
time
clock ID 333
clock ticks 331
time-accounting
information 3557
user ID
effective 851, 2888, 2934, 2957
real 1001, 2934, 2957
saved 2957
user name, getting 901, 903
yielding 2743
zombies, preventing children from becoming 2992, 3027, 3162
processor affinity 3492
Processor Version Register 3391
procmgr_daemon() 2136
PROCGR_DAEMON_KEEPMASK 2136
PROCGR_DAEMON_NOCHDIR 2136
PROCGR_DAEMON_NOCLOSE 2136
PROCGR_DAEMON_NODEVNULL 2136
PROCGR_EVENT_DAEMON_DEATH 2138
procmgr_event_notify() 2139
PROCGR_EVENT_SYNC 2138
procmgr_event_trigger() 2143
procmgr_guardian() 2145
procmgr_session() 2150
PROCGR_SESSION_SETPGRP 2150
PROCGR_SESSION_SETSID 2150
PROCGR_SESSION_SIGNAL_LEADER 2150
PROCGR_SESSION_SIGNAL_PGRP 2150
PROCGR_SESSION_SIGNAL_PID 2150
PROCGR_SESSION_TCSSETSID 2150
__programe 2152
program entry function 1604
PROT_EXEC 1308, 1703, 1712, 1748
PROT_NOCACHE 1308, 1703, 1712, 1748
PROT_NONE 1308, 1703, 1712, 1748
Protocol Data Unit (PDU) See SNMP
protocols
database
closing 552
entries, getting 936, 938, 940
entry structure 2153
opening 2928
ICMP (Internet Control Message Protocol) 1066
ICMP6 (Internet Control Message Protocol v6) 1068
INET6 (Internet protocol v6 family) 1127
interprocess communication 3628
IP (Internet Protocol) 1416
IP6 (Internet Protocol v6) 1442
IPsec (secure Internet Protocol) 1423
TCP (Transmission Control Protocol) 3432
UDP (User Datagram Protocol) 3599
protoent 2153
PROT_READ 1308, 1703, 1712, 1748
PROT_WRITE 1308, 1703, 1712, 1748
proxy server (SOCKS) 3823
pseudo-random numbers
double 491, 565
int 2460, 2462
long
  nonnegative 1581, 1963
  signed 1518, 1783, 2464
  seed, setting 2820, 3219, 3221, 3223
sequence, initializing 3221
state
  initializing 1161, 1530
  switching 2953
pseudo-ttys
  opening 731, 1991
  preparing for a login 731, 1576
pthread_attr_t 2254
PTHREAD_COND_INITIALIZER 2229
pthread_abort() 2154
PTHREAD_ABORTED 2154
pthread_atfork() 2156
pthread_attr_destroy() 2158
pthread_attr_getdetachstate() 2160
pthread_attr_getguardsize() 2162
pthread_attr_getinheritsched() 2164
pthread_attr_getschedparam() 2166
pthread_attr_getschedpolicy() 2168
pthread_attr_getscope() 2170
pthread_attr_getstackaddr() 2172
pthread_attr_getstacklaz() 2174
pthread_attr_getstacksize() 2178
pthread_attr_init() 2180
pthread_attr_setdetachstate() 2183
pthread_attr_setguardsize() 2185
pthread_attr_setinheritsched() 2188
pthread_attr_setschedparam() 2190
pthread_attr_setschedpolicy() 2192
pthread_attr_setscope() 2194
pthread_attr_setstackaddr() 2196
pthread_attr_setstacklaz() 2198
pthread_attr_setstacksize() 2202
pthread_barrierattr_destroy() 2210
pthread_barrierattr_getpshared() 2212
pthread_barrierattr_init() 2214
pthread_barrierattr_setpshared() 2216
pthread_barrier_destroy() 2204
pthread_barrier_init() 2206
PTHREAD_BARRIER_INITIALIZER() 2206

pthread_barrier_wait() 2208

PTHREAD_CANCEL 3480

pthread_cancel() 2218

PTHREAD_CANCELASYNCHRONOUS 2256, 2365, 3486

2367, 3487

PTHREAD_CANCELDEFERRED 2256, 2367, 3487

PTHREAD_CANCELDISABLE 2256, 2365

PTHREAD_CANCELED 2275, 2408

PTHREAD_CANCELENABLE 2256, 2365, 3487

pthread_cleanup_pop() 2220

pthread_cleanup_push() 2222

pthread_condrattr_destroy() 2242

pthread_condrattr_getclock() 2244

pthread_condrattr_getpshared() 2246

pthread_condrattr_init() 2248

pthread_condrattr_setclock() 2250

pthread_condrattr_setpshared() 2252

pthread_cond_broadcast() 2225

pthread_cond_destroy() 2227

pthread_cond_init() 2229

PTHREAD_COND_INITIALIZER 3341, 3344

pthread_cond_signal() 2232

pthread_cond_timedwait() 2234

pthread_cond_wait() 2238

pthread_create() 2255

PTHREAD_CREATEDETACHED 2183, 3486

PTHREAD_CREATEJOINABLE 2180, 2183, 3486

PTHREAD_DESTRUCTOR_ITERATIONS 2278

pthread_detach() 2259

pthread_equal() 2261

pthread_exit() 2263

PTHREAD_EXPLICIT_SCHED 2188, 2190, 2192, 3485, 3486

pthread_getconcurrency() 2265

pthread_getcpuclockid() 2267

pthread_getschedparam() 2271

pthread_getspecific() 2273

PTHREAD_INHERIT_SCHED 2180, 2188

pthread_join() 2275

pthread_key_create() 2277

pthread_key_delete() 2281

pthread_kill() 2283

PTHREAD_MULTISIG_ALLOW 2256, 3486

PTHREAD_MULTISIG_DISALLOW 2256, 3486

pthread_mutexattr_destroy() 2304

pthread_mutexattr_getprioceiling() 2306

pthread_mutexattr_getprotocol() 2308

pthread_mutexattr_getpshared() 2311

pthread_mutexattr_getrecursive() 2313

pthread_mutexattr_gettype() 2315

pthread_mutexattr_init() 2318

pthread_mutexattr_setprioceiling() 2321

pthread_mutexattr_setprotocol() 2324
pthread_mutexattr_setpshared() 2326

pthread_mutexattr_setrecursive() 2327

pthread_mutexattr_settype() 2330

PTHREAD_MUTEX_DEFAULT 2330

pthread_mutex_destroy() 2285

PTHREAD_MUTEX_ERRORCHECK 2329

pthread_mutex_getprioceiling() 2287

pthread_mutex_init() 2289

PTHREAD_MUTEX_INITIALIZER 2289

pthread_mutex_lock() 2291

PTHREAD_MUTEX_NORMAL 2329

PTHREAD_MUTEX_RECURSIVE 2329

pthread_mutex_setprioceiling() 2295

pthread_mutex_timedlock() 2297

pthread_mutex_trylock() 2300

pthread_mutex_unlock() 2302

pthread_once() 2332

PTHREAD_ONCE_INIT 2332

PTHREAD_PRIO_INHERIT 2318, 2337, 3371

PTHREAD_PRIO_NONE 2323

PTHREAD_PRIO_PROTECT 2323, 3371

PTHREAD_PROCESS_PRIVATE 2216, 2248, 2325, 2362, 2397

PTHREAD_PROCESS_SHARED 2216, 2248, 2252, 2325, 2362, 2397

PTHREAD_RECURSIVE_DISABLE 2313, 2318, 2327

PTHREAD_RECURSIVE_ENABLE 2313, 2327

PTHREAD_RWLOCK_INITIALIZER 2289

pthread_rwlockattr_destroy() 2356

pthread_rwlockattr_getpshared() 2358

pthread_rwlockattr_init() 2360

pthread_rwlockattr_setpshared() 2362

pthread_rwlock_destroy() 2335

pthread_rwlock_init() 2337

PTHREAD_RWLOCK_INITIALIZER 2337

pthread_rwlock_rdlock() 2340

pthread_rwlock_timedrdlock() 2342

pthread_rwlock_timedwrlock() 2345

pthread_rwlock_tryrdlock() 2348

pthread_rwlock_trywrlock() 2350

pthread_rwlock_unlock() 2352

pthread_rwlock_wrlock() 2354

PTHREAD_SCOPE_PROCESS 3486

PTHREAD_SCOPE_SYSTEM 2180, 2194, 3486

pthread_self() 2364

pthread_setcancelstate() 2365

pthread_setcanceltype() 2367

pthread_setconcurrency() 2369

pthread_setschedparam() 2374

pthread_setspecific() 2375

pthread_sigmask() 2377

pthread_sleepon_broadcast() 2379

pthread_sleepon_lock() 2381

pthread_sleepon_signal() 2383

pthread_sleepon_timedwait() 2385

pthread_sleepon_unlock() 2389

pthread_sleepon_wait() 2391

pthread_spin_destroy() 2395

pthread_spin_init() 2397

pthread_spin_lock() 2400

pthread_spin_trylock() 2402

pthread_spin_unlock() 2404
Index

PTHREAD_STACK_LAZY 2198
PTHREAD_STACK_MIN 2196,
      2197, 2202, 2203, 3485,
      3489
PTHREAD_STACK_NOTLAZY 2198
pthread_testcancel()  2406
pthread_timedjoin()  2407
_PTR_BITS_  113
pulse_attach()  2413
_PULSE_CODE_COIDDEATH  302,
      2410
_PULSE_CODE_DISCONNECT  303,
      2410
_PULSE_CODE_MAXAVAL  1850,
      2410, 3006
_PULSE_CODE_MINAVAL  1850,
      2410, 3006
_PULSE_CODE_NET_ACK  2410
_PULSE_CODE_NET_DETACH  2410
_PULSE_CODE_NET_UNBLOCK  2410
_PULSE_CODE_THREADDEATH  304,
      460, 2410
_PULSE_CODE_UNBLOCK  304,
      2410
pulse_detach()  2416

Q

   _QNX_  113
QNX 4 classification  107
QNX Neutrino classification  107
qnx_crypt()  2448
   _QNXNTO_  113
QoS (Quality of Service)  1946
qsort()  2451
qtime  3393
Quality of Service (QoS)  1946
QUERY  2552
quick sort  2451
quotients
  integer 468
  long integer 1534

R
Raccept() 2455
radix-independent exponents 1573, 2699, 2702
raise() 1578, 2457
rand() 2460
RAND_MAX 2460, 2462
random numbers
  double 491, 565
  int 2460, 2462
  long
    nonnegative 1581, 1963
    signed 1518, 1783, 2464
    seed, setting 2820, 3219, 3221, 3223
    sequence, initializing 3221
    state
      initializing 1161, 1530
      switching 2953
random() 2464
rand_r() 2462
raw input mode
  buffer 745, 2008
  conditions for input request 2493
  FORWARD qualifier 2494
  MIN qualifier 2493
  TIME qualifier 2493
  TIMEOUT qualifier 2494
Rbind() 2467
rcmd() 2469, 3824
Rconnect() 2472
rcvid (receive identifier) 1819, 1831
  checking validity of 1878
rdchk() 2474
read() 1236, 2480, 2492
  resource managers,
    implementing in 1357, 1360
read-write locks
  attributes
    creating 2360
    destroying 2356
    process-shared 2358, 2362
    destroying 2335, 3352
    initializing 2337, 3372
    locking
      for reading 2340, 2342, 2348
      for writing 2345, 2350, 2354
    unlocking 2352
readblock() 2489
readcond() 2492
readdir() 1981, 2498
readdir_r() 2502
readlink() 2505
  resource managers,
    implementing in 1363
read_main_config_file() 2485
readv() 2508
realloc() 2512
realpath() 2515
realtime timers
  busy-waiting 1920, 1926, 1928, 1930
  calibrating 1923
creating 3508
destroying 3512
expiry status 3514
overruns 3516
setting 3546
threads 341
time
going 3518
setting 3521
rebooting 3389
rebooting on bankruptcy 2752
receive identifier See rcvid
RECEIVED_MESSAGE 3131
receiving
messages 1818, 1830
from a socket 2518, 2521, 2525
pulses 1824, 1827
\texttt{re\_comp()} 2476
\texttt{recv()} 2518
\texttt{recvfrom()} 2521
\texttt{recvmsg()} 2525
\texttt{re\_exec()} 2478
\texttt{regcomp()} 2529
\texttt{regerror()} 2533
\textit{regex\_t} 2528
\texttt{regexexec()} 2536
\texttt{REG\_EXTENDED} 2528
\texttt{regfree()} 2538
\texttt{REG\_ICASE} 2528
regulators
devices
access to, gaining and relinquishing 1710, 1897
reading 2068, 2071, 2074, 2077
writing 2081, 2084, 2086, 2088
floating-point exceptions 736
Machine Status Register 3391
Processor Version
Register 3391
real-mode software
interrupts 1208
TSC (Time Stamp Counter) 350
\texttt{regmatch\_t} 2536
\texttt{REG\_NEWLINE} 2528
\texttt{REG\_NOSUB} 2528, 2537
\texttt{REG\_NOTBOL} 2535
\texttt{REG\_NOTEOL} 2535
regular expressions
basic 2529
compiling 2476, 2529
errors, explaining 2533
extended 2530
freeing 2538
string, comparing to 2478, 2536
\texttt{remainder()}, \texttt{remainderf()} 2540
remainders
floating point 493, 2540
integer 468
long integer 1534
remote hosts
commands, executing on 2469, 2660
identity, checking 2690
\texttt{remove()} 2542
\texttt{rename()} 2545
\texttt{RES\_DEBUG} 2548
\texttt{RES\_DEFNAMES} 2549, 2561
\texttt{RES\_DNSRCH} 2549, 2561

November 2, 2006
Index 3959
residue, floating point  715
RES_INIT  2549
res_init()  2548
resmgr_attr_t  2569
resmgr_connect_funcs_t  2578
resmgr_context_t  2585
resmgr_io_funcs_t  2605
resmgr_attach()  2568
resmgr_block()  2575
_RESMGR_CONNECT_NFUNCS  1280
resmgr_context_alloc()  2580
resmgr_context_free()  2583
_RESMGR_DEFAULT  1388
resmgr_detach()  2587
_RESMGR_DETACH_ALL  2587
_RESMGR_DETACH_PATHNAME  2587
resmgr_devino()  2591
RESMGR_FLAG_AFTER  1740, 1742
_RESMGR_FLAG_AFTER  2570
RESMGR_FLAG_ATTACH_OTHERFUNC  2570
RESMGR_FLAG_BEFORE  1739, 1742
_RESMGR_FLAG_BEFORE  2570
_RESMGR_FLAG_DIR  2571
_RESMGR_FLAG_FTYPEONLY  2571
_RESMGR_FLAG_OPAQUE  2572
RESMGR_FLAG_OPAQUE  1740, 1742
_RESMGR_FLAG_SELF  2572
_resmgr_handle_grow()  2594
resmgr_handler()  2599
resmgr_handle_func()  2596
_resmgr_io_func()  2602
resmgr_iofuncs()  2609
_RESMGR_IO_NFUNCS  1280
resmgr_msgread()  2611
resmgr_msgreadv()  2613
resmgr_msgwrite()  2615
resmgr_msgwritev()  2617
_RESMGR_NOREPLY  1388
_RESMGR_NPARTS()  2619
_resmgr_ocb()  2621
resmgr_open_bind()  2623
resmgr_pathname()  2626
_RESMGR_PATHNAME_LOCALPATH  2626
_RESMGR_PTR()  2629
_RESMGR_STATUS()  2631
resmgr_unbind()  2633
res_mkquery()  2552
resolv.conf, contents of  374
resolver routines
errors  1030, 1036, 1045
initializing  2548
Internet domain names
compressing  487
expanding  489
options  2548
queries  2552, 2555, 2558, 2561, 2563
resource database manager
about  2666
device numbers
attaching  2681
detaching  2685
resources
creating  2673
destroying  2677
querying  2688
reserving  2666
returning  2679
resource managers
access, checking 1243  
arming for notification 1410  
attaching 2568  
attributes  
  initializing 1228  
  locking 1230, 1235, 1238, 1296  
  structure 1233  
  time members,  
    updating 1386  
  unlocking 1240  
clients  
  information about 1256  
  unblocking 1388, 1390  
connect functions 2578  
  default values, setting 1280  
  open 1343  
connect messages  
  file type reply 1219  
  link reply 1221  
  structure 1212  
connection IDs 383  
context  
  allocating 2580  
  freeing 2583  
  structure 2585  
database, expanding  
  capacity 2594  
detaching 2587  
device number, getting 2591  
device-control commands 427  
file descriptors, mapping to  
  OCBs 2596  
function tables,  
  initializing 1280  
helper functions  
  chmod 1246  
  chown 1251  
  close 1258  
  devctl 1267  
  fdinfo 1274  
  link 1284  
  lseek 1298  
  mknod 1304  
  mmap 1306  
  open 1338  
  openfd 1346  
  pathconf 1352  
  read 1360  
  readlink 1363  
  rename 1367  
  space 1371  
  stat 1374  
  sync 1383  
  unlink 1394  
  utime 1398  
  write 1407  
I/O functions  
  chmod 1249  
  chown 1254  
  client connection, getting  
    for 2609  
  close 1261  
  default values, setting 1280  
  devctl 1271  
  fdinfo 1276  
  lock 1291  
  lseek 1301  
  mmap 1310  
  OCB, close 1265  
  OCB, lock 1296  
  OCB, unlock 1396  
  openfd 1350  
  pathconf 1355
Index

read 1357
retrieving 2602
stat 1376
structure 2605
sync 1381
unblock 1390
utime 1401
write 1404
inode number, getting 2591
iov_t
filling 2629
getting 2619
locks (not implemented) 1287, 1289, 1294
messages
blocking while waiting
for 2575
handling 2599
reading 2611, 2613
writing 2615, 2617
notification
arming for 1410
installing, polling, and
removing 1314
removing for a client 1319
triggering 1321
Open Control Block (OCB)
allocation 1326
attaching 1325, 2623
detaching 1263, 1329, 2633
file descriptors, mapping
to 2596
freeing 1332
getting 2621
structure 1335
path
attaching to 2568
detaching from 2587
getting 2626
server attributes, getting 1226
status, returning 2631
synchronization, checking to see
if required 1379
threads in 1235
resources, system
creating 2673
destroying 2677
limits
getting 845, 956
setting 2938
querying 2688
reserving 2666
returning 2679
usage, getting 959
res_query() 2555
res_querydomain() 2558
RES_RECURSE 2549
res_search() 2561
res_send() 2563
RES_STAYOPEN 2549
RES_USEVC 2549
rewind() 2635
rewinddir() 2638
\texttt{rftp} 3823
Rgetsockname() 2641
rindex() 2643
rint(), rintf() 2645
RLIM_INFINITY 2938, 2942
RLIMIT_AS 2938
RLIMIT_CORE 2938
RLIMIT_CPU 2938
RLIMIT_DATA 2938
RLIMIT_FSIZE 2938, 2942
RLIMIT_MEMLOCK 2938

3962  Index  November 2, 2006
Index

RLIMIT_NOFILE 845, 2938, 2942
RLIMIT_NPROC 2938
RLIMIT_NTHR 2938
RLIMIT_OFILE 2938
RLIMIT_RSS 2938
RLIMIT_STACK 2938
RLIMIT_VMEM 2938
RLIM_SAVED_CUR 2942
RLIM_SAVED_MAX 2942
Rlisten() 2648
rmdir() 2650
R_OK 126, 516
root directory, changing 320
roots
cube 276
square 3217
round-robin scheduling 2192
rounding
floating point 2645
integers 278, 711
mode, floating-point 742
ROUTE (system packet forwarding database) 2653
Rrcmd() 2660
rresvport() 2662
Rselect() 2665
rshd 2469
rsrcrequest_t 2673
rsrcrequest_t 2667
rsrcrequest_t 2667
rsrcrequest_t 2666
rsrcrequest_t 2673
rsrcrequest_t 2677
rsrcrequest_t 2679
rsrcrequest_t 2681
rsrcrequest_t 2685
RSRCDMGR_DMA_CHANNEL 2668, 2674, 2687
RSRCDMGR_FLAG_ALIGN 2668
RSRCDMGR_FLAG_NOREMOVE 2674
RSRCDMGR_FLAG_RANGE 2668
RSRCDMGR_FLAG_RSVP 2674
RSRCDMGR_FLAG_SHARE 2668
RSRCDMGR_FLAG_TOPDOWN 2668
RSRCDMGR_IO_PORT 2668, 2674, 2687
RSRCDMGR_IRQ 2668, 2674, 2687
RSRCDMGR_MEMORY 2668, 2674, 2687
RSRCDMGR_PCI_MEMORY 2668, 2674, 2687
rsrcrequest_t 2688
RSRCMGR_IRQ 2674, 2687
rt_metrics 2656
rt_message 2656
RTA_AUTHOR 2657
RTA_BRD 2657
RTA_DST 2657
RTA_GATEWAY 2657
RTA_GENMASK 2657
RTA_IFA 2657
RTA_IFP 2657
RTA_NETMASK 2657
rtelnet 3823
RTF_BLACKHOLE 2657
RTF_CLONING 2657
RTF_DONE 2657
RTF_DYNAMIC 2657
RTF_GATEWAY 2657
RTF_HOST 2657
RTF_LLINFO 2657
RTF_MASK 2657
RTF_MODIFIED 2657
RTF_PROTO1 2657

November 2, 2006
Index

© 2006, QNX Software Systems GmbH & Co. KG.

RTF_PROTO2 2657
RTF_REJECT 2657
RTF_STATIC 2657
RTF_UP 2657
RTF_XRESOLVE 2657
RTLD_DEFAULT 484
RTLD_GLOBAL 480
RTLD_GROUP 481
RTLD.LAZY 480
RTLD_LOCAL 480
RTLD_NOW 480
RTLD_WORLD 481
RTM_ADD 2655
RTM_CHANGE 2655
RTM_DELADDR 2655, 2656
RTM_DELETE 2655
RTM_GET 2655
RTM_IFINFO 2655, 2656
RTM_LOSING 2655
RTM_MISS 2655
RTM_NEWADDR 2655, 2656
RTM_REDIRECT 2655
RTM_RESOLVE 2655
RTV_EXPIRE 2657
RTV_HOPCOUNT 2657
RTV_MTU 2657
RTV_RPIPE 2657
RTV_RTT 2657
RTV_RTTVAR 2657
RTV_SPIPE 2657
RTV_SSTRESH 2657
RUN_LVL 1005, 3644
rusage 959
ruserok() 2690

S

SAB_NOCLDSTOP 3020, 3026
SAB_ONSTACK 585
SAB_SIGINFO 1548, 2991, 3027, 3058
SAT (System Analysis Toolkit) 1193, 3582
sbrk() 2696
scalb(), scalbf() 2699
scalbn(), scalbnf() 2702
scalloc() 2705
scandir() 2708
scanf() 2709
_SC_ARG_MAX 3375
_SC_CHILD_MAX 3375
_SC_CLK_TCK 3375
_SC_GETGR_R_SIZE_MAX 860
_SC_GETPW_R_SIZE_MAX 947, 952, 987, 991
sched_aps_clear_crit_parms 2768
sched_aps_createParms 2754
sched_aps_events_parm 2771
sched_aps_info 2747
sched_aps_join_parms 2759
sched_aps_lookup_parms 2759
sched_aps_mark_crit_parms 2767
sched_aps_modify_parms 2761
sched_aps_overall_stats 2765
sched_aps_parms 2753
sched_aps_partition_info 2757
sched_aps_partition_stats 2763
sched_aps_query_process_parms 2777
sched_aps_query_thread_parms 2769
sched_aps_security_parms 2773
sched_param 2729
SCHED_ADJTOHEAD 2782
SCHED_ADJTOTAIL 2782
SCHED_APS_ADD_SECURITY 2773
SCHED_APS_ATTACH_EVENTS 2771
SCHED_APS_BNKR_BASIC 2752
SCHED_APS_BNKR_CANCEL_BUDGET 2752
SCHED_APS_BNKR_LOG (not implemented) 2752
SCHED_APS_BNKR_REBOOT 2752
SCHED_APS_BNKR_RECOMMENDED 2752
SCHED_APS_CLEAR_CRITICAL 2768
SCHED_APS_CREATE_PARTITION 2754
SCHED_APS_JOIN_PARTITION 2759
SCHED_APS_LOOKUP 2759
SCHED_APS_MARK_CRITICAL 2767
SCHED_APS_MODIFY_PARTITION 2761
SCHED_APS_OVERALL_STATS 2765
SCHED_APS_PARTITION_STATS 2763
SCHED_APS_PSTATS_IS_BANKRUPT_NOW 2765
SCHED_APS_PSTATS_WAS_BANKRUPT 2765
SCHED_APS_QUERY_PARMS 2747
SCHED_APS_QUERY_PARTITION 2757
SCHED_APS_QUERY_PROCESS 2777
SCHED_APS_QUERY_THREAD 2769
SCHED_APS_SCHEDPOL_BMP_SAFETY 2750
SCHED_APS_SCHEDPOL_FREETIME_BY_RATIO 2749
SCHED_APS_SEC_BASIC 2775
SCHED_APS_SEC_FLEXIBLE 2775
SCHED_APS_SEC_JOIN_SELF_ONLY 2777
SCHED_APS_SEC_LOCK_PARTITIONS 2774
SCHED_APS_SEC_NONZERO_BUDGETS 2776
SCHED_APS_SEC_PARENT_JOINS 2776
SCHED_APS_SEC_PARENT_MODIFIES 2776
SCHED_APS_SEC_PARTITIONS_LOCKED 2777
SCHED_APS_SEC_RECOMMENDED 2774
SCHED_APS_SEC_ROOT0_OVERALL 2775
SCHED_APS_SEC_ROOT_JOINS 2776
SCHED_APS_SEC_ROOT MAKES_CRITICAL 2776
SCHED_APS_SEC_ROOT MAKES_PARTITIONS 2775
SCHED_APS_SEC_SYS_JOINS 2776
SCHED_APS_SEC_SYS MAKES_CRITICAL 2776
SCHED_APS_SEC_SYS MAKES_PARTITIONS 2776
SCHED_APS_SEC_SYSTEM MAKES_PARTITIONS 2776
SCHED_APS_SET_PARMS 2753
SchedCtl(), SchedCtl_f() 2746
SCHED_FIFO 2192, 2723, 2725, 2740, 2782, 2784, 3164
SchedGet(), SchedGet_f() 2781
sched_getparam()  2718
sched_get_priority_adjust()  2721
sched_get_priority_max()  2723
sched_get_priority_min()  2725
sched_getscheduler()  2727
_schedinfo(), _schedinfo_r()  2785
SCHED_NOCHANGE  2192, 3485
SCHED_OTHER  2192, 2723, 2725, 2740, 2782, 2784, 3164
SCHED_RR  2192, 2723, 2725, 2740, 2782, 2784, 3164
_sched_rr_get_interval()  2735
_schedset(), _schedset_r()  2788
sched_setparam()  2737
sched_setscheduler()  2741
SCHED_SPORADIC  2192, 2782, 3164

scheduling
  information, getting  2785
  parameters  2729
  threads, getting for  2166, 2190, 2271, 2781
  threads, inheriting  2164, 2188, 3486
  threads, setting for  2374, 2788, 3162

policy
  don’t change  2192
  FIFO  2192
  other  2192
  processes, getting for  2727
  processes, setting for  2741, 3162
  round-robin  2192
  sporadic  2192
  threads, getting for  2168, 2271, 2781
  threads, inheriting  2164, 2188, 3486
  threads, setting for  2192, 2374, 2788, 3485
  yielding  2791

_schedyield()  2743
_SchedYield(), _SchedYield_r()  2791
_SC_JOB_CONTROL  3376
SCM_RIGHTS  3629
_SC_GROUPS_MAX  3375
scoped addresses  1128
_SC_OPEN_MAX  3376
_SC_PAGESIZE  2185
scripts, running  605, 628, 3158, 3162, 3169, 3174, 3179, 3184, 3190, 3196, 3201, 3206, 3210
_SC_SAVED_IDS  3376

SCTP
  associations
    adding or removing
    addresses  2795
    branching into a separate
    socket  2808
    locally bound addresses  2800, 2804
    messages
      receiving  2810
      sending  2813
      multihomed endpoints,
      connecting to  2798
      peer addresses  2802, 2806
      protocol  2793

SCTP (Stream Control Transmission Protocol)
  classification  107

sctp_bindx()  2795
SC_VERSION 3376
searchenv() 2817
secure Internet Protocol See IPsec
secure RPC domain 374
Security Policy Database
(SPD) 1427
security, setting 2773, 2774
seed48() 2820
SEEK_CUR 663, 777, 1298, 1372, 1586, 1596
seekdir() 2822
SEEK_END 663, 777, 1298, 1372, 1586, 1596
SEEK_SET 662, 663, 777, 1298, 1372, 1586, 1596
segments
data
maximum size 2938
data, end of 519, 522
text
beginning of 253
end of 583
SELECT_FLAGExcept 2831
SELECT_FLAG_READ 2831
SELECT_FLAG_REARM 2831
SELECT_FLAG_SRVEXCEPT 2832
SELECT_FLAG_WRITE 2831
select_query() 2838
semaphores
named
accessing and creating 2850
closing 2840
destroying 2861
posting 2854, 3366
unnamed
destroying 2842, 3352
initializing 2846, 3372
value
decrementing 2856, 2859,
2863, 3368
getting 2844
incrementing 2854, 3366
setting 2846, 2850
waiting on 2863, 3368
with a time limit 2856
without blocking 2859
sem_close() 2840
sem_destroy() 2842
sem_getvalue() 2844
sem_init() 2846
sem_open() 2850
sem_post() 2854
sem_timedwait() 2856
sem_trywait() 2859
sem_unlink() 2861
SEM_VALUE_MAX 2846, 2847, 2850
sem_wait() 2863
send() 2866
sendmsg() 2869
sendto() 2872
serial number, getting 374
serial ports
  opening 1716
  reading 1722
  script, running on 1726
  writing 1733
servent 2874
server attributes, getting 1226
servers
  connections
    creating 172, 300
    destroying 174, 307
    information about 387, 395
data server
  applications, registering and
deregistering 500, 506
  variables, creating and
destroying 495, 498
  variables, flags 502
  variables, getting and
  setting 504, 508
PCI
  attaching 2039
  detaching 2051
services
  database
    closing 556
    entries, getting 966, 968,
    970
    entry structure 2874
    opening 2944
sessions
  character device terminal
    drivers, support for 2150
    controlling terminal 3423
  creating 1576, 2946
  current 2624
disassociating 633
ID, getting 972
leader, creating 2924, 3163
membership, inheriting 586,
  593, 601, 611, 617, 623,
  3190
remote node 3161
system daemons 2136
termination, notification
  of 2138
setbuf() 2875
setbuffer() 2877
setdomainname() 2880
setegid() 2882
setenv() 606, 629, 2885, 3169,
  3179, 3196, 3206
seteuid() 2888
setgid() 2891
setgrent() 2894
setgroups() 2896
sethostent() 874, 880, 2898
sethostname() 2900
SETIOV() 2902
setitimer() 2904
setjmp() 2907
setkey() 2910
setlinebuf() 2912
setlocale() 2915, 3328
setlogmask() 2917
setnetent() 2919
setpgid() 2921
setpgrp() 2924
setprio() 2926
setprotoent() 2928
setpwent() 2930
setregid() 2931
SET_REQ_MSG 3117
setreuid() 2934
setrlimit(), setrlimit64() 2938
setservent() 2944
setuid() 2946
setsockopt() 2950
setspent() 2952
setstate() 2953
settimeofday() 2955
setutent() 2957

not honoring on mounted filesystems 1739

setutent() 2960

setvbuf() 2962

sfree() 2965

sh 2105

shadow password database
   closing 557
   entries
      reading 690, 987, 992
      structure 2432
      writing 2432
      rewinding 2952

shared locks 707

shared memory
   access protection,
      changing 1748
   attributes, modifying 2968, 2977
   mapping 1704
   opening 2980
   removing 2986
   unmapping 1895

shared objects
   addresses, translating 471
   closing 473

debugging 482
errors 475
opening 477
symbol, getting address of 484

SH_COMPAT 1217, 1348, 3150, 3154
SH_DENYNO 1217, 1348, 3150, 3154
SH_DENYRD 1217, 1348, 3150, 3154
SH_DENYRW 1217, 1348, 3150, 3154
SH_DENYWR 1217, 1348, 3150, 3154

SHELL 326, 2105, 3398
shell scripts, running 605, 628, 3158, 3162, 3169, 3174, 3179, 3184, 3190, 3196, 3201, 3206, 3210

shm_ctl() 2968

SHMCTL_ANON 2967, 2976
SHMCTL_GLOBAL 2967, 2976
SHMCTL.LAZYWRITE 2967, 2976
SHMCTL_LOWERPROT 2967, 2976
SHMCTL_PHYS 2967, 2976
SHMCTL_PRIV 2967, 2976

shm_ctl_special() 2977
shm_open() 2980
shm_unlink() 2986
shutdown() 2988

SLASYNCIO 2993, 3028, 3034
side channels 382
S_IEEXEC 1216, 3232
S_IFBLK 1216, 1222, 3233
S_IFCHR 1216, 1222, 3233
S_IFDIR 1216, 1222, 1685, 3233
S_IFIFO 1216, 1222, 1685, 3233
S_IFLNK 1216, 1222, 3233
S_IFMT 1216, 1222, 3233
example of use 1464
S_IFNAM 1216, 1222, 3233
S_IFREG 1216, 1222, 3233
S_IFSOCK 1216, 1222, 1464, 3233
SIGABRT 119, 3024

sigaction 3026
sigaction() 2990
sigaddset() 2996
SIGALRM 149, 899, 2905, 3025, 3532
process, sending to 3509, 3535, 3596
SIG_BLOCK 3055
sigblock() 2998
SIGBUS 1705, 3025
SIGCHLD 633, 959, 2991, 3020, 3021, 3025, 3718
default actions 2991, 3027
ignoring 2992, 3027
SIGCONT 633, 2998, 3025
default actions 2992
sigdelset() 3000
SIG_DFL 585, 2991, 3019
sigemptyset() 3002
SIGEMT 3024
SIG_ERR 3021
SIGEV_CLEAR_CRITICAL() 3009

sigevent 1412, 1788, 3004
bankruptcy 2751, 2771
critical 2751, 2767
system overload 2771
type, determining with
SIGEV_GET_TYPE() 3005
SIGEV_GET_TYPE() 3005
SIGEV_INTR 1171, 1177, 1205, 1789, 3004
SIGEV_INTR_INIT() 3005
SIGEV_MAKE_CRITICAL() 3009
SIGEV_NONE 1548, 2771, 3004
SIGEV_NONE_INIT() 2771, 3005
SIGEV_PULSE 1171, 1177, 1758, 1789, 3004, 3509
SIGEV_PULSE_INIT() 3006
SIGEV_PULSE_PRIO_INHERIT 3006
SIGEV_SIGNAL 1172, 1177, 1548, 1758, 1789, 3004, 3509
SIGEV_SIGNAL_CODE 1172, 1177, 1758, 3004, 3509
SIGEV_SIGNAL_CODE_INIT() 3007
SIGEV_SIGNAL_INIT() 3006
SIGEV_SIGNALTHREAD 1172, 1177, 1758, 3004, 3509
SIGEV_SIGNALTHREAD_INIT() 3007
SIGEV_SIGNAL_VALUE_INIT() 3006
SIGEV_THREAD 3004
SIGEV_THREAD_INIT() 3008
SIGEV_UNBLOCK 1789, 3004
SIGEV_UNBLOCK_INIT() 3008
sigfillset() 3011
SIGFPE 3025
SIG_HOLD 3162
SIG_HUP 633, 3024
process groups, targeting 633
SIG_IGN 585, 2992, 3019, 3027, 3163
SIGILL 3024

siginfo_t 2992, 3028, 3034
SIGINT 3024
process groups, sending to 3441
SIG_INTR 2751
SIGIO 3025
  default actions 2991, 3027
SIGIOT 3024
sigismember() 3013
SIGKILL 2378, 2993, 2998, 3020, 3021, 3025
siglongjmp() 3015
sigmask() 3017
_SIGMAX 3026
_SIGMIN 3026
sign, copying 398
signal() 3019
SignalAction(), SignalAction_r() 3024
SignalKill(), SignalKill_r() 3032
SignalProcmask(),
  SignalProcmask_r() 3038
signals
  actions 2990, 3019, 3024
    default 2991
  blocking 2998, 3063
    SIGCONT 2998
    SIGKILL 2378, 2998
    SIGSTOP 2378, 2998
    SIGTTOU 757, 759, 2442, 2444
catching
    SIGKILL 3020
    SIGSTOP 3020
ignoring 2992
SIGCHLD 959, 3021
SIGKILL 2993, 3021
SIGSTOP 2993, 3021
SIGTTOU 757, 759, 2442, 2444
information about 3075
masks
  constructing 3017
  restoring 3015
  saving 3061
  signal-blocked 3038
  threads 2377
names 3024
POSIX 3024
process groups, targeting 633, 1520, 1523, 2993, 3029, 3032
processes
  pending 3053
  queuing 2991, 3058
  suspending until delivered 3042
sending 1520, 1523, 2457, 3032
SIGABRT 119
SIGALRM 149, 899, 2905
SIGBUS 1705
SIGCHLD 633, 2991, 3020
SIGCONT 633
SIGHOLD 3162
SIGHUP 633
SIGKILL 3020
SIGPIPE 757, 759, 980, 1732, 1734, 2442, 2444, 2446
SIGSEGV 1177, 1895, 1899, 2185, 2938
SIGTRAP 418
SIGXCPU 2938
SIGXFSZ 2938
sets
  adding to 2996
  initializing 3002, 3011
membership, checking for 3013
removing from 3000
string describing 3304
threads
mask 3055, 3065
threads, targeting 2283, 2993, 3029, 3032
unblocking 3071
user-defined 3025
waiting for 2020, 3045, 3051, 3065, 3068, 3073, 3075
_signalstub() 3023
SignalSuspend(), SignalSuspend_r() 3042
SignalWaitinfo(), SignalWaitinfo_r() 3045
significand(), significandf() 3048
sigpause() 3051
sigpending() 3053
SIGPIPE 757, 759, 980, 1732, 1734, 2442, 2444, 2446, 3025
SIGPOLL 3025
sigprocmask() 3055
SIGPWR 3025
sigqueue() 3058
SIGQUIT 3024
SIGRTMAX 3026
SIGRTMIN 3026
SIGSEGV 1177, 1201, 1203, 1895, 1899, 2185, 2938, 3025
sigsetjmp() 3061
SIG_SETMASK 3055
sigsetmask() 3063
SIGSTOP 2378, 2993, 2998, 3020, 3021, 3025
default actions 2992
__sigsuspend() 3065
SIGSYS 3025
SIGTERM 3025
sigtimedwait() 3068
SIGTRAP 418, 3024
SIGTSTP 3025
SIGTTIN 3025, 3668
SIGTTOU 757, 759, 2442, 2444, 3025, 3668
SIG_UNBLOCK 3055
sigunblock() 3071
SIGURG 3025
default actions 2991, 3027
process groups, sending to 3142
SIGUSR1 3025
SIGUSR2 3025
sigwait() 3073
sigwaitinfo() 3075
SIGWINCH 3025
default actions 2991, 3027
SIGXCPU 2938
SIGXFSZ 2938, 3585
SL_MAXA VAIL 3007
SL_MSGQ 2993, 3028, 3034
SL_MINA VAIL 3007
Simple Network Management Protocol See SNMP
sin(), sinf() 3077
sines 3077
hyperbolic 3079
inverse hyperbolic 166
sinh(), sinhf() 3079
SIOCGIFCONF 1419
SIOCGIFFLAGS 1419
SL_QUEUE 2993, 3028, 3034
S_IREAD 1216, 3232
INDEX

S_IRGRP 3231, 3604
S_IROTH 3231, 3604
S_IRUSR 3231, 3604
S_IRWXG 2849, 3231, 3604
S_IRWXO 2849, 3231, 3604
S_IRWXU 2849, 3231, 3604
S_ISBLK() 782, 3233
S_ISCHR() 782, 3233
S_ISDIR() 782, 3233
S_ISFIFO() 782, 3233
S_ISGID 313, 314, 318, 650, 653, 783, 1528, 3232, 3585
S_ISLNK() 783, 3233
S_ISNAM() 3233
S_ISREG() 783, 3233
S_ISSOCK() 3234
S_ISUID 313, 314, 318, 650, 653, 783, 1528, 3232, 3585
S_ISVTX 313
site-local addresses 1128
SI_TIMER 2993, 3028, 3034
SI_USER 2993, 3028, 3034
S_IWGRP 3231, 3604
S_IWOTH 3231, 3604
S_IWRITE 1216, 3232
S_IWUSR 3231, 3604
S_IXGRP 3231, 3604
S_IXOTH 3231, 3604
S_IXUSR 3231, 3604
sleep() 3081
sleeping
  for microseconds 3636
  for milliseconds 423, 1933, 1935
sleepon locks
  destroying 3085
  initializing 3087
locking 2381, 3089
unblocking 2379, 2383, 3083, 3091
unlocking 2389, 3093
waiting 2385, 2391, 3095
_sleepon_broadcast() 3083
_sleepon_destroy() 3085
_sleepon_init() 3087
_sleepon_lock() 3089
_sleepon_signal() 3091
_sleepon_unlock() 3093
_sleepon_wait() 3095
slogb() 3097
_SLOG_CRITICAL 3100
_SLOG_DEBUG1 3100
_SLOG_DEBUG2 3100
_SLOG_ERROR 3100
slogf() 3100
slogi() 3103
_SLOG_INFO 3100
_SLOG_NOTICE 3100
_SLOG_SETCODE() 3099, 3684
_SLOG_SHUTDOWN 3100
_SLOG_WARNING 3100
_smalloc() 3105
SNMP (Simple Network Management Protocol)
  classification 107
  daemon, configuration file for 2485
messages
  creating 3118
  freeing 3109
  reading 3120
  sending 3126
Protocol Data Unit (PDU)
  creating 3118
freeing 3109
processing 3120
sending 3126
structure 3113
sessions
  characteristics 3129
  closing 3107
  opening 3111
timeouts, handling 3133
transactions,
asynchronous 3123

\texttt{snmp} 3113

\texttt{snmp_session} 3111, 3129

\texttt{snmp_close()} 3107

\texttt{SNMPCONFIGFILE} 2487

\texttt{snmpd.conf} 2485

\texttt{snmpd.conf} 2485

\texttt{SNMP_DEFAULT_ADDRESS} 3130

\texttt{SNMP_DEFAULT_COMMUNITY_LENGTH} 3129

\texttt{SNMP_DEFAULT_ENTERPRISE_LENGTH} 3114

\texttt{SNMP_DEFAULT_ERRINDEX} 3114

\texttt{SNMP_DEFAULT_ERRSTAT} 3114

\texttt{SNMP_DEFAULT_PEERNAME} 3130

\texttt{SNMP_DEFAULT_REMPORT} 3130

\texttt{SNMP_DEFAULT_REQID} 3114

\texttt{SNMP_DEFAULT_RETRIES} 3129

\texttt{SNMP_DEFAULT_TIME} 3115

\texttt{SNMP_DEFAULT_TIMEOUT} 3129

\texttt{SNMPERR_BAD_ADDRESS} 3111, 3127

\texttt{SNMPERR_BAD_LOCPORT} 3112

\texttt{SNMPERR_BAD_SESSION} 3107, 3127

\texttt{SNMPERR_GENERR} 3112, 3118, 3127

\texttt{snmp_errno} 3107, 3111, 3118, 3127

\texttt{snmp_free_pdu()} 3109

\texttt{snmp_open()} 3111

\texttt{snmp_pdu_create()} 3118

\texttt{snmp_read()} 3120
  using with \texttt{select()} 3123

\texttt{snmp_select_info()} 3123
  using with \texttt{select()} 3123

\texttt{snmp_send()} 3126

\texttt{snmp_timeout()} 3133
  using with \texttt{select()} 3123

\texttt{SNMP_VERSION.1} 3113, 3131

\texttt{SNMP_VERSION.2} 3113, 3131

\texttt{snprintf()} 3135

\texttt{SO_BINDTODEVICE} 979

\texttt{SO_BROADCAST} 979

\texttt{sockaddr_un} 3628

\texttt{sockatmark()} 3138

\texttt{sockcred} 3630

\texttt{SOCK_CREDSIZE()} 3630

\texttt{SOCK_DGRAM} 378, 983, 1127, 1416, 1445, 3140, 3141, 3599, 3628, 3823

\texttt{socket()} 1066, 1068, 1127, 1416, 1423, 1442, 2653, 2793, 3141, 3432, 3599, 3628

\texttt{socketpair()} 3144

sockets
  addresses
    errors 820
    freeing 765
    getting 825
    structure 133
connections
  accepting on 123, 2455
  initiating 378, 1937, 2472
listening for 1552, 2648
shutting down 2988
status 1940
creating 3141
a pair of 3144
connected 3144
datagrams 3140, 3141, 3599
debugging 979
file descriptors, testing for
association 1464
manager, getting and setting
information about 3379
messages
peeking at 2517, 2520, 2524
receiving from 2518, 2521,
2525
sending to 2866, 2869, 2872
names
binding to 242, 2467
getting 974, 2641
options
getting 977
setting 2950
out-of-band (OOB) mark 3138
privileged IP port, binding
to 245
privileged ports, getting 2662
raw 3140, 3141
stream 3140, 3141
types 3140
determining 980, 984
SOCKET 976, 2655
SO_DEBUG 979
SO_DONTROUTE 979
SO_ERROR 980
SO_KEEPALIVE 980
SO_LINGER 976, 980, 2949
SOCKET 976, 2655
SO_OOBINLINE 981
sopen() 3150
sopenfd() 3155
SO_RCVBUFSIZE 981
SO_RCVLOWAT 981
SO_RCVTIMEO 976, 982, 2949
SO_REUSEADDR 982
SO_REUSEPORT 982
sorting
directory entries 155, 2708
quick sort 2451
SO_SNDBUFSIZE 981
SO_SNDELAY 983
SO_SNDTIMEO 976, 983, 2949
SO_TIMESTAMP 983
SO_TYPE 984

commands, executing
remotely 2660
compiling for 3823
initializing 3147
library 3824
sockets
connections 2455, 2472,
2648
names 2467, 2641

socks3r.lib 3823
SOCKSinit() 3147, 3823
SOCK_STREAM 123, 378, 1127,
1416, 1445, 1553, 2455,
2470, 3140, 3141, 3432,
3628
SO_DEBUG 979
SO_DONTROUTE 979
SO_ERROR 980
SO_KEEPALIVE 980
SO_LINGER 976, 980, 2949
SOCKET 976, 2655
SO_OOBINLINE 981
sopen() 3150
sopenfd() 3155
SO_RCVBUFSIZE 981
SO_RCVLOWAT 981
SO_RCVTIMEO 976, 982, 2949
SO_REUSEADDR 982
SO_REUSEPORT 982
sorting
directory entries 155, 2708
quick sort 2451
SO_SNDBUFSIZE 981
SO_SNDELAY 983
SO_SNDTIMEO 976, 983, 2949
SO_TIMESTAMP 983
SO_TYPE 984
Index

SO_USELOOPBACK 984, 2655
space, amount free for a filesystem 787
space, filesystem 3239
space, testing a character for 1478, 1501
spawn() 3158
spawn* family of functions 49, 57, 1981
SPAWN_ALIGN_DEFAULT 3161
SPAWN_ALIGN_FAULT 3161
SPAWN_ALIGN_MASK 3163
SPAWN_ALIGN_NOFAULT 3161
SPAWN_CHECK_SCRIPT 3162
SPAWN_DEBUG 3162
SPAWN_EXEC 3162
SPAWN_EXPLICIT_SCHED 3162, 3164, 3165
SPAWN_FDCLOSED 3157, 3188
SPAWN_HOLD 3162
spawnl() 3169
spawnle() 3174
spawnlp() 3179
spawnlpe() 3184
SPAWN_NEWPGROUP 3163
SPAWN_NOZOMBIE 3162
spawnp() 3189
SPAWN_SEARCH_PATH 3162
SPAWN_SETGROUP 3159, 3162, 3163, 3190
SPAWN_SETND 3162
SPAWN_SETSID 3163
SPAWN_SETSIGDEF 3160, 3163, 3164, 3191
SPAWN_SETSIGIGN 3163, 3164
SPAWN_SETSIGMASK 3160, 3163, 3164, 3191
SPAWN_SETSTACKMAX 3163, 3164
SPAWN_TCSETPGROUP 3163
spawn() 3196
spawnve() 3201
spawnvp() 3206
spawnvpe() 3210
SPD (Security Policy Database) 1427
special characters 1733
spinlocks
destroying 2395
initializing 2397
locking 2400, 2402
unlocking 2404
sporadic scheduling 2192
sprintf() 3214
spwd 2432
sqrt(), sqrtf() 3217
square roots 3217
srand() 3219
srand48() 3221
srandom() 3223
_srealloc() 3225
sscanf() 3228
SS_REPL_MAX 2731
stack
maximum size 2938, 3163
memory, allocating from 152
overflow, protecting
against 2185
threads 2172, 2174, 2178, 2196, 2198, 2202
stat 782, 3230
stat(), stat64() 1590, 3230

November 2, 2006
resource managers, implementing in 1374, 1376
STATE_CONDVAR 3346, 3550
STATE_INTR 3550
STATE_JOIN 3550
STATE_MUTEX 3345, 3346, 3550
STATE_RECEIVE 3550
STATE_REPLY 1842, 3550
STATE_SEM 3550
STATE_SEND 1842, 3550
STATE_SIGSUSPEND 3550
STATE_SIGWAITINFO 3550
st_atime 1679, 1682, 2095, 2482, 2510
statfs, statvfs64 786, 3238
statfs(), statvfs64() 3238
st_ctime 318, 651, 654, 1528, 1543, 1679, 1682, 2095, 2650, 3804
stderr 69, 658, 698, 2092, 3242
buffering 2877, 2912
command-line options, errors when parsing 917
daemons 2136
formatted messages on 571, 3665, 3701, 3723
host errors 1036
STDERR_FILENO 698, 2094, 2105, 3243
STDOUT_FILENO 698, 2094, 2105, 3244
stdout 69, 658, 698, 2430, 2443
buffering 2877, 2912
characters, writing 2126, 3679, 3703, 3800
daemons 2136
output, formatted 2126, 3679, 3703, 3800
strings, writing 2430
wide characters, writing 2443
STDOUT_FILENO 698, 2094, 2105, 3244
st_atime 1682, 2095
st_mode 3233
st_mtime 1543, 1679, 1682, 2095, 2650, 3804
ST_NOSUID 586, 787, 3161, 3239
straddstr() 3245
strcasecmp() 3247
strcat() 3250
strchr() 3252
strcmpl() 3256
strcoll() 3258, 3328
strcp() 3260
strcspn() 3262
ST_RDONLY 787, 3239
strdup() 3264
Stream Control Transmission Protocol See SCTP
stream I/O
buffering
associating with 2875, 2962
block 2877
line 2912
characters
Index

pushing back 3624, 3626
reading 681, 683, 687, 832, 834, 836, 838, 1013, 1015
writing 750, 752, 2419, 2421, 2423, 2425, 2441, 2443
closing 656, 658
end-of-file 674
clearing 329
errors 676
clearing 329
messages, printing 2092
file descriptors
associating streams with 671
getting 698
files
flushing 678, 713
locking 709, 801
opening 722
reading 1164
unlocking 806
input, formatted 775, 818, 2709, 3672, 3677, 3681, 3705, 3813
output, formatted 748, 813, 2126, 3669, 3675, 3679, 3703, 3800
pipes
closing 2090
creating 2094
opening 2104
position
getting 685, 792
setting 778, 780
reading 760
reopening 769
rewinding 2635
seeking 778
strings
reading 964
writing 754, 758, 2430
telling 792
temporary files 1689, 1691, 3566
wide characters
orientation 811
reading 693, 695
writing 756
words
getting next 1011
writing 2439
writing 815
stream sockets 3140, 3141
stream, returning to remote command 2469, 2660
streams
flushing 3416
strerror() 3266
strftime() 3268
stricmp() 3274
strings See also characters; wide characters
character, filling with 3292, 3302
comparing 238, 1650, 1657
case-insensitive 3247, 3256, 3274, 3280, 3290
case-sensitive 3254, 3285
locale’s collating sequence, using 3258
concatenating 3245, 3250, 3283
configuration, getting and setting 373
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copying</td>
<td>240, 1645, 3260, 3264, 3287, 3328</td>
</tr>
<tr>
<td>encrypting</td>
<td>408, 520, 2448, 2910</td>
</tr>
<tr>
<td>error messages</td>
<td>2092, 3266</td>
</tr>
<tr>
<td>formatted</td>
<td>3135, 3214, 3686, 3689</td>
</tr>
<tr>
<td>hexadecimal numbers, converting to/from</td>
<td>207</td>
</tr>
<tr>
<td>input, formatted</td>
<td>775, 2709, 3672, 3681</td>
</tr>
<tr>
<td>IP addresses, converting to/from</td>
<td>1119, 1122</td>
</tr>
<tr>
<td>IPv4 addresses, converting to/from</td>
<td>1097, 1099, 1114, 1116</td>
</tr>
<tr>
<td>length</td>
<td>3276</td>
</tr>
<tr>
<td>lowercase, converting to</td>
<td>3278</td>
</tr>
<tr>
<td>matching</td>
<td>719</td>
</tr>
<tr>
<td>network numbers, converting to/from</td>
<td>1112</td>
</tr>
<tr>
<td>node descriptors, converting to/from</td>
<td>1945, 1952</td>
</tr>
<tr>
<td>numbers, converting to/from</td>
<td>205, 207, 209, 211, 1509, 1594, 3310, 3314, 3321, 3324, 3601, 3648</td>
</tr>
<tr>
<td>output, formatted</td>
<td>748, 2126, 3669, 3679</td>
</tr>
<tr>
<td>paths, resolving</td>
<td>2515</td>
</tr>
<tr>
<td>reversing</td>
<td>3298</td>
</tr>
<tr>
<td>scanning input from</td>
<td>3228, 3692, 3697</td>
</tr>
<tr>
<td>searching characters</td>
<td>1095, 2643, 3252, 3296</td>
</tr>
<tr>
<td>sets of characters</td>
<td>3262, 3294, 3306</td>
</tr>
<tr>
<td>sets of wide characters</td>
<td>3737, 3749, 3755</td>
</tr>
<tr>
<td>slashes (/)</td>
<td>235, 440</td>
</tr>
<tr>
<td>strings</td>
<td>3308</td>
</tr>
<tr>
<td>wide characters</td>
<td>3729, 3751</td>
</tr>
<tr>
<td>signal descriptions</td>
<td>3304</td>
</tr>
<tr>
<td>splitting</td>
<td>3300, 3315, 3318</td>
</tr>
<tr>
<td>stdout, writing to</td>
<td>2430</td>
</tr>
<tr>
<td>streams reading from</td>
<td>687</td>
</tr>
<tr>
<td>writing</td>
<td>754, 758</td>
</tr>
<tr>
<td>substrings comparing, case-insensitive</td>
<td>3280, 3290</td>
</tr>
<tr>
<td>comparing, case-sensitive</td>
<td>3285</td>
</tr>
<tr>
<td>time_t, converting to/from</td>
<td>412</td>
</tr>
<tr>
<td>time, formatted</td>
<td>3268, 3739</td>
</tr>
<tr>
<td>tm, converting to/from</td>
<td>161</td>
</tr>
<tr>
<td>tokens, splitting into</td>
<td>3315, 3318</td>
</tr>
<tr>
<td>transforming</td>
<td>3328</td>
</tr>
<tr>
<td>uppercase, converting to</td>
<td>3326</td>
</tr>
<tr>
<td>zeroing</td>
<td>256</td>
</tr>
<tr>
<td>strlen()</td>
<td>3276</td>
</tr>
<tr>
<td>strlwr()</td>
<td>3278</td>
</tr>
<tr>
<td>strncasecmp()</td>
<td>3280</td>
</tr>
<tr>
<td>strncat()</td>
<td>3283</td>
</tr>
<tr>
<td>strncmp()</td>
<td>3285, 3328</td>
</tr>
<tr>
<td>strncpy()</td>
<td>3287, 3328</td>
</tr>
<tr>
<td>strncpy()</td>
<td>3287, 3328</td>
</tr>
<tr>
<td>strnicmp()</td>
<td>3290</td>
</tr>
<tr>
<td>strnset()</td>
<td>3292</td>
</tr>
</tbody>
</table>
**Index**

© 2006, QNX Software Systems GmbH & Co. KG.

```
strpbrk() 3294
strchr() 3296
strrev() 3298
strsep() 3300
strsignal() 3302
strlen() 3304
strspn() 3306
strstr() 3308
strtok() 3310
strtoimax() 3314
strtok() 3315
strtok_r() 3318
strstr() 3318
strtol(), strtoll() 3321
strtoul(), strtoull() 3324
stroumax() 3314
struct, offset of members within 1971
strupr() 3326
strxfrm() 3328
S_TYPEISMQ() 783, 3234
S_TYPEISSEM() 783, 3234
S_TYPEISSHM() 783, 3234
suboptions, parsing 994
SUN_LEN() 3628
swab() 3331
swprintf() 3333
swscanf() 3335
symbolic links
creating 3337
deleting 2542, 3631
information, getting 1590
ownership, changing 1527
reading 2505
resolving 2515
temporary 2016, 2018
symlink() 3337
SYMLOOP_MAX 2506
sync() 3340
resource managers,
implementing in 1381, 1383
SyncCondvarSignal(),
SyncCondvarSignal_r() 3341
SyncCondvarWait(),
SyncCondvarWait_r() 3344
SyncCtl(), SyncCtl_r() 3350
SyncDestroy(), SyncDestroy_r() 3352
synchronization objects See also
mutexes; semaphores; threads
creating 3372
destroying 3352
mutexes
events 3350, 3355
locking 3358, 3363
priority 3350
reviving 3361
semaphores
incrementing 3366, 3368
threads
blocking 3344
waking up 3341
SyncMutexEvent(),
SyncMutexEvent_r() 3355
SyncMutexLock(),
SyncMutexLock_r() 3358
SyncMutexRevive(),
SyncMutexRevive_r() 3361
```
SyncMutexUnlock(),
    SyncMutexUnlock_r() 3363
SyncSemPost(), SyncSemPost_r() 3366
SyncSemWait(), SyncSemWait_r() 3368
SyncTypeCreate(),
    SyncTypeCreate_r() 3372
sysconf() 3375
sysctl() 3379
sys_errlist 575
syslog() 3386
sysmgr_reboot() 3389
sys_nerr 575
sys_nsig 3024
SYSPAGE_CPU_ENTRY() 3391
SYSPAGE_ENTRY() 3393
    qtime 3393
    boot_time 3393
    cycles_per_sec 3393
    .syspage_ptr 3396
sys_siglist 3024
system
    clock
        getting 360
        period, getting and
        setting 357
        setting 360
        ticks per second 3393
    commands, executing 3397
    daemons 415, 2136
        termination, notification of 2138
    events
        notification of 2139
        triggering 2143
    hardware information 1052,
        1055, 1056, 1058
    instruction set architecture 373
    limits, getting 3375
    rebooting 3389
    rebooting on bankruptcy 2752
    resources
        creating 2673
        destroying 2677
        limits, getting 845, 956
        limits, setting 2938
        querying 2688
        reserving 2666
        returning 2679
        usage, getting 959
    time
        since booting 3393
        time, adjusting 348
    System Analysis Toolkit
        (SAT) 1193, 3582
    system databases
        groups
            closing 523
            ID, getting information about 858, 860
            membership 1159
            name, getting information about 863, 865
            next entry, getting 855
            rewinding 2894
        passwords
            closing 554
            encrypting 408, 2448
            entry, getting for a user 945, 948, 950, 953
            entry, getting next 942
            rewinding 2930
        shadow passwords
Index

closing 557
entry, reading 690, 987, 992
entry, structure 2432
entry, writing 2432
rewinding 2952
system message log
   closing 368
   log priority mask 2917
   opening 1989
   writing to 3386, 3699
      blocks 3097
      formatted output 3100, 3685
      integers 3103
system packet forwarding database
   See ROUTE
system page 3396
   CPU-specific entry, getting a pointer to 3391
   entry, getting a pointer to 3393
System partition 2754
system() 49, 3397

tcgetpgrp() 3421
tcgetsid() 3423
tcgetsiz() 3425
TCIFLUSH 3416
tcinject() 3427
TCIOFF 3414
TCIOFFHW 3414
TCIOFLUSH 3416
TCION 3414
TCIONHW 3414
tciscars() 3430
TCOFLUSH 3416
TCOFF 3413
TCOFFHW 3413
TCOON 3413
TCOONHW 3413
TCP (Transmission Control Protocol) 3432
   connection, closing 525
   SOCKS 3824
TCP/IP
   address information
      addrinfo 133
   addresses
      manipulating 1103
      network numbers,
         extracting 1108
      strings, converting
         to/from 1097, 1099, 1114,
            1116, 1119, 1122
   errors 1030, 1036, 1045
   host entries
      getting 879, 883, 885, 888
   hosts database
      opening 2898
   Internet domain names
      compressing 487

tan(), tanff() 3404
tangents 3404
   hyperbolic 3406
   inverse hyperbolic 200
tanh(), tanhff() 3406
target operating system 113
tcdrain() 3408
tcdropline() 3410
tcflow() 3413
tcflush() 3416
tcgetattr() 3419
expanding 489
messages
receiving 2518, 2521, 2525
sending 2866, 2869, 2872
network database
closing 551
opening 2919
protocols database
closing 552
opening 2928
services database
closing 556
entry structure 2874
opening 2944
sockets
ports, binding to 245
TCP_KEEPALIVE 3433
TCP_MAXSEG 3433
TCP_NODELAY 984, 3433
TCSADRAIN 3438
TCSAFLUSH 3438
TCSANOW 3438
tcsendbreak() 3435
tcsetattr() 3438
tcsetpgrp() 3441
tcsetsid() 3444
tcsetsize() 3446
tell() 1588
tell(), tell64() 3448
telldir() 3451
tempnam() 3453
temporary files
creating 3566
creating and opening 1689
name, generating 1691, 3453, 3569
TERM 326
terminal control
characters, injecting 3427
communications line
break condition,
asserting 3435
disconnecting 3410
draining 3408
flow control 3413
flushing 3416
process group ID 3423
getting 3421, 3441
size 3425, 3446
terminal group, starting new 3163
terminals
attributes, setting 290
canonical input buffer 745, 2008
control
attributes 3419, 3438
structure 3456
controlling
making 3444
path name 410
file descriptor, testing for
association with 1458
input speed 280, 294
operating attributes 1224
output speed 283, 297
raw input buffer 745, 2008
reading 2492
TERMINFO 326
termios 290, 2492, 3438, 3456
text segment
beginning of 253
end of 583
ThreadCancel(), ThreadCancel_r() 3479
Index

ThreadCreate(), ThreadCreate_r() 3484
ThreadCtl(), ThreadCtl_r() 1168, 1177, 1182, 1185, 1187, 1195, 1198, 1201, 1203, 3490
ThreadDestroy(), ThreadDestroy_r() 3498
ThreadDetach(), ThreadDetach_r() 3501
ThreadJoin(), ThreadJoin_r() 3503
thread_pool_attr() 3464
thread_pool_control() 3461
THREAD_POOL_CONTROL_HIWATER 3460
THREAD_POOL_CONTROL_INCREMENT 3460
THREAD_POOL_CONTROL_LOWATER 3460
THREAD_POOL_CONTROL_MAXIMUM 3461
THREAD_POOL_CONTROL_NONBLOCK 3461
thread_pool_create() 3463
thread_pool_destroy() 3470
thread_pool_limits() 3474
thread_pool_start() 3476
threads
aborting 2154
adaptive partitioning 2746
attributes 3484
contention scope 2170, 2194
destroying 2158
detach state 2160, 2183
guard area, size of 2162, 2185
initializing 2180
scheduling parameters 2166, 2190
scheduling policy 2168, 2192
stack address 2172, 2196
stack size 2178, 2202
stack, lazy 2174, 2198
bankruptcy, notifying of 2771
barriers
attributes 2206, 2210, 2214
attributes,
process-shared 2212, 2216
destroying 2204
initializing 2206
waiting at 2208
blocking 3344
busy-waiting 1920, 1926, 1928, 1930
calibrating 1923
canceling 2218, 3479
cancellation
cleanup handlers 2220, 2222
points 3487
points, creating 2406
state 2365
type 2367
clock ID 2267
concurrency 2265, 2369
condition variables
attributes 2242, 2248
attributes, clock 2244, 2250
attributes,
process-shared 2246, 2252
blocking on 2234, 2238
destroying 2227, 3352
initializing 2229, 3372
unblocking 2225, 2232
creating 2255, 3484
critical
determining if 2769
marking as 2767, 3009
data 2273, 2277, 2281, 2375, 3487
destroying 3498
detached 2183, 3486
detaching from a
process 2259, 3501
errno 575, 3487
events
attaching 2771
files, locking 709
fork handlers, registering 2156
freezing 3494
I/O privileges, requesting 3492
IDs
calling thread 2364
comparing 2261
initializing 2332
joinable 2183, 3486
joining 2275, 3503
with a time limit 2407
keys 2277, 2281
local storage 2273, 2277, 2281, 2375, 3487
maximum number of 2938
misaligned access
response 3490
mutexes
attributes, destroying 2304
attributes, initializing 2318
attributes, priority
ceiling 2306, 2321
attributes,
process-shared 2311,
2326
attributes, recursive 2313,
2327
attributes, scheduling
protocol 2308, 2324
attributes, type 2315, 2330
destroying 2285, 3352
initializing 2285, 3372
locking 2291, 2297, 2300
priority ceiling 2287, 2295
unlocking 2302
noncritical, marking as 2768, 3009
once-initialization 2332
partitions
determining 2769
joining 2759, 2776
pool See also resource managers
attributes, changing 3461,
3474
creating 3463
destroying 3470
starting 3476
priority on receiving a
pulse 3006
private data 2273, 2277, 2281,
2375, 3487
processor affinity 3492
read-write locks
attributes, creating 2360
attributes, destroying 2356
attributes,
process-shared 2358,
2362
Index

destroying  2335, 3352
initializing  2397
locking  2400, 2402
unlocking  2404
return status  3485
scheduling parameters  2271, 2374, 2781, 2788
scheduling policy  2271, 2374, 2781, 2788, 3485
inheriting  2164, 2188, 3486
scope  3486
signal mask
getting  3055
restoring  3015
saving  3061
setting  3055, 3065
signal-blocked  3038
signals
initial state  3487
mask  2377
sending  2283
targeting  2993, 3029, 3032
terminating on  3486
waiting for  2020
sleep on locks
destroying  3085
initializing  3087
locking  2381, 3089
unblocking  2379, 2383, 3083, 3091
unlocking  2389, 3093
waiting  2385, 2391, 3095
spinlocks
destroying  2395
initializing  2337, 3372
locking for reading  2340, 2342, 2348
locking for writing  2345, 2350, 2354
unlocking  2352
return status  3485
scheduling parameters  2271, 2374, 2781, 2788
scheduling policy  2271, 2374, 2781, 2788, 3485
inheriting  2164, 2188, 3486
scope  3486
signal mask
getting  3055
restoring  3015
saving  3061
setting  3055, 3065
signal-blocked  3038
signals
initial state  3487
mask  2377
sending  2283
targeting  2993, 3029, 3032
terminating on  3486
waiting for  2020
sleep on locks
destroying  3085
initializing  3087
locking  2381, 3089
unblocking  2379, 2383, 3083, 3091
unlocking  2389, 3093
waiting  2385, 2391, 3095
spinlocks
destroying  2395
initializing  2397
locking  2400, 2402
unlocking  2404
stack  3485
suspending  341, 1918, 3065, 3081
synchronizing  2208
system overload, notifying of  2771
terminating  2263
terminating unconditionally  2154
unfreezing  3494
waking up  2379, 2383, 3083, 3091, 3341
yielding  2743, 2791
zombies  3486
tick size, changing  357, 2748
ticksize, getting and setting  357
time_t  3506
tm, converting to/from  1026, 1028
time members
in attribute structure of resource managers  1237
Time Stamp Counter (TSC)  350
time to live (TTL)  1416
time zone
abbreviations  3592
default  374
offset from UTC  3563
setting  3593
time()  3506
timeb  795
TIMED_OUT  3131, 3133
timeout, setting on a blocking state  3524
timeouts, SNMP 3133
TIMER_ABSTIME 340, 3520
TimerAlarm(), TimerAlarm r() 3532
timer_create() 3508
TimerCreate(), TimerCreate r() 3535
timer_delete() 3512
TimerDestroy(), TimerDestroy r() 3538
timer_getexpstatus() 3514
timer_getoverrun() 3516
timer_gettime() 3518
TimerInfo(), TimerInfo r() 3542

TimerTimeout(), TimerTimeout r() 3550
timer_timeout(), timer_timeout r() 3524
times
  booting, since 3393
calendar
    current 3506
    local, converting
      to/from 1559, 1561, 1693
    structure 3564
clock
  adjusting 348
  cycles 350
  getting 337
  getting and setting 360
  ID, getting 333, 353
  period, getting and
    setting 357
  resolution, getting 335
  setting 344
current
  calendar 3506
  getting 795, 999
  setting 2955
daylight saving time 417, 3593
difference, calculating 435
files
  access 1386, 1398, 1401, 3641
  modification 808, 1386, 1398, 1401, 3638, 3641
  status-change 1386, 1398, 1401
  formatting 2915, 3268, 3739
  local
calendar, converting to/from 1559, 1561, 1693
nanoseconds
timespec, converting to/from 1965, 3561
processes
   execution time limit, getting 2735
   execution time, in clock ticks 331
specification structure 3560
time_t
   strings, converting to/from 1026, 1028
   timespec, converting to/from 1026, 1028
   timespec
      nanoseconds, converting to/from 1965, 3561
   tm
      strings, converting to/from 161
times() 3557
timespec 335, 3560
   nanoseconds, converting to/from 1965, 3561
timespec2nsec() 3561
timezone 3563, 3593
_TLS 3487
tm 1026, 1559, 1693, 3564, 3739
   strings, converting to/from 161, 412
   time_t, converting to/from 1026, 1028
tmpfile() 3566
   TMP_MAX 3453
tmpnam() 3569
tms 3557
tokens, breaking a string into 3315, 3318, 3764
tolower() 3572
TOS (type of service) 1416
TOSTOP 3458
toupper() 3574
towctrans() 3576
towlower() 3578
towupper() 3580
T_PTR 2551, 2554, 2557, 2560
TraceEvent() 3582
Transmission Control Protocol  See TCP
trigonometry  See also hyperbolic functions
   arccosine 129
   arcsine 164
   arctangent 196, 198
   cosine 400
   sine 3077
tangent 3404
TRP2_REQ_MSG 3117
TRP_REQ_MSG 3117
truncate() 3585
TRY_AGAIN 1031, 1037
TSC (Time Stamp Counter) 350
TTL (time to live) 1416
ttyname() 3588
tyyname_r() 3590
type of service (TOS) 1416
typographical conventions liii
TZ 326, 3593
tzname 3592, 3593
tzset() 1693, 3271, 3593
U

ualarm() 3596
UDP (User Datagram Protocol) 3599
   not supported by SOCKS 3823
UIO_MAXIOV 2508, 2511, 3810
ulltoa() 3601
ultoa() 3601
umask() 3604
umount() 3607
UNALIGNED_PUT16() 3609
UNALIGNED_PUT32() 3611
UNALIGNED_PUT64() 3613
UNALIGNED_RET16() 3615
UNALIGNED_RET32() 3617
UNALIGNED_RET64() 3619
uname() 3621
ungetc() 3624
ungetwc() 3626
Unicode 98
union, offset of members within 1971
Unix classification 107
UNIX-domain protocol 3628
unlink() 3631
unsetenv() 3634
uppercase
   characters, converting to 3574
   strings, converting to 3326
   testing a character for 1481, 1503
   wide characters, converting to 3576, 3580, 3782
usage of system resources 959
User Datagram Protocol (UDP) 3599

not supported by SOCKS 3823
user information file
   closing 559
   entry 3644
   reading 1003, 1008
   renaming 3646
   returning to beginning of 2960
   searching 1005
   writing 2436
USER_PROCESS 1005, 3645
users
   IDs
      effective 851, 2888, 2934, 2957
      real 1001, 2934, 2957
      saved 2957
      set-user 3376
   names 901, 903
   password database, getting entry for 945, 948, 950, 953
   processes, maximum per real user ID 2938, 3375
usleep() 3636
utilities, locating 374
utimbuf 808, 3638
utime() 3638
   resource managers, implementing in 1398, 1401
utimes() 3641
utmp 3644
utmpname() 3646
utoa() 3648
utsname 3621
V

va_arg() 3651
va_copy() 3657
va_end() 3659
valloc() 3663

variable_list 3115
variable-length argument lists
(“varargs”) 3651, 3657, 3659, 3661
coercion 3652
variables, global
_ambksiz 157
_argc 159
_argv 160
_auxv 234
_btext 253
daylight 417
_edata 519
_end 522
erno 574
_etext 583
optarg 916
opterr 917
optind 916
optopt 917
__progname 2152
stderr 3242
stdin 3243
stdout 3244
sys_errlist 575
sys_nerr 575
sys_nsig 3024
__syspage_ptr 3396
sys_siglist 3024
timezone 3563
tzname 3592

va_start() 3661
verr(), verrx() 3665
vfork() 3667
vfprintf() 3669
vfscanf() 3672
vfvprintf() 3675
vfvwscanf() 3677
video memory, sharing 1707
virtual 8086 mode 1208
void pointers, size of 113
vprintf() 3679
vwscanf() 3681
vslogf() 3685
vsnprintff() 3686
vsprintf() 3689
vsscanf() 3692
vswprintf() 3695
vswscanf() 3697
vsyslog() 3699
vwarn(), vwarnx() 3701
vwprintf() 3703
vwscanf() 3705

W

wait() 3558, 3707, 3714, 3721
wait3() 3711
wait4() 3714
waitid() 3718
waitpid() 3558, 3721
warn(), warnx() 3723
warnings, formatted on stderr 3701, 3723
WCONTINUED 3710, 3713, 3717, 3720
WCOREDUMP() 3708
wrtomb() 3725
wcsat() 3727
wcschr() 3729
wcsncmp() 3731
wcsoll() 3733, 3775
wcsncpy() 3735
wcsespn() 3737
wcsftime() 3739
wcslen() 3741
wcsnca() 3743
wcsncmp() 3745
wcsncpy() 3747
wcsxbrk() 3749
wcschr() 3751
wcsrtombs() 3753
wesspn() 3755
wesstr() 3757
wcstod() 3759
wcstof() 3759
westotmax() 3763
wcstok() 3764
wcstoll() 3767
wcstold() 3759
wcstoll() 3767
wcstombs() 3769
wcstayl(), wcstou() 3773
wcstaylmax() 3763
wcstb() 3777
wcstomb() 3779
wcstrans() 3782
wcstod() 3784
WEXITED 3710, 3713, 3717, 3720
WEXITSTATUS() 3398, 3708
whitespace, testing a character for 1478, 1501

wide characters See also
characters, strings
classes 3784
converting 3576, 3782
copying 3790
overlapping objects 3792
lowercase, converting to 3576, 3578, 3782
multibyte characters, converting
to/from 1623, 1633, 3725, 3779
conversion object, status of 1626
searching
in a string 3729, 3751
in memory 3786
sets of, searching for 3737, 3749, 3755
setting 3794
single-byte characters,
converting to/from 254, 3777
stdin, reading from 1015
stdout, writing to 2443
streams
orientation 811
pushing back 3626
reading 693, 695, 1013
writing to 756, 2441
strings
comparing 3731, 3733, 3745, 3788
concatenating 3727, 3743
copying 3735, 3747, 3775
formatted 3333, 3695
input, formatted 818, 3335, 3677, 3705, 3813
length 3741
multibyte characters,
converting to/from 1628, 1630, 3753, 3769
numbers, converting
to/from 3759, 3763, 3767, 3773
output, formatted 813, 3675, 3703, 3800
searching for a set of wide
characters 3737, 3749, 3755
searching for a string 3757
searching for a wide
character 3729, 3751
splitting into tokens 3764
streams, writing to 758
transforming 3775
testing for
alphabetic 1485
alphanumeric 1483
character class 1489
control character 1487
decimal digit 1491
hexadecimal digit 1505
lowercase 1495
printable 1493, 1497
punctuation 1499
uppercase 1503
whitespace 1501
uppercase, converting to 3576, 3580, 3782
wmemcmp() 3788
wmemcpy() 3790
wmemmove() 3792
wmemset() 3794
WNOHANG 3710, 3713, 3717, 3720
WNOWAIT 3710, 3713, 3717, 3720
W_OK 126, 516
word expansions 3796, 3798
wordexp() 3796
wordfree() 3798
working directory 310, 840, 1017
wprintf() 3800
WRDE_NOSYS 3796
write() 1236, 3802
resource managers,
implementing in 1404
writeblock() 3807
writerv() 3810
wscanf() 3813
WSTOPPED 3710, 3714, 3717, 3721
WSTOPSIG() 3708
WTERMSIG() 3708
WUNTRACED 3710, 3714, 3717, 3721
X
X_OK 126, 516
Y
y0(), y0f() 3815
y1(), y1f()  3817
yn(), ynf()  3819

Z

zombies
  preventing children from becoming  2992, 3027, 3162
  threads  3486