QNX® Momentics® DDK

Character Devices

For targets running QNX® Neutrino® 6.3 or later

Printed under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: **info@qnx.com**
Web: **http://www.qnx.com/**

Publishing history

Electronic edition published 2005.

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (**www.qnx.com**). You'll find a wide range of support options, including our free online support site, the Developer Support Center.

Contents

List of Figures

About the Character DDK

What you'll find in this guide

The following table may help you find information quickly:

For information about:	See this chapter:
The character I/O system	Character I/O Architecture
The 8250 serial driver	8250 serial driver
Functions provided by the **`io-char`** library	Character I/O Library

Assumptions

To use this guide, you need to have:

- sufficient hardware documentation for your hardware in order to be able to program all the registers
- a working knowledge of the C programming language.

Building DDKs

You can compile the DDK from the IDE or the command line.

- To compile the DDK from the IDE:

  Please refer to the Managing Source Code chapter, and "QNX Source Package" in the Common Wizards Reference chapter of the *IDE User's Guide*.

- To compile the DDK from the command line:

  Please refer to the release notes or the installation notes for information on the location of the DDK archives.

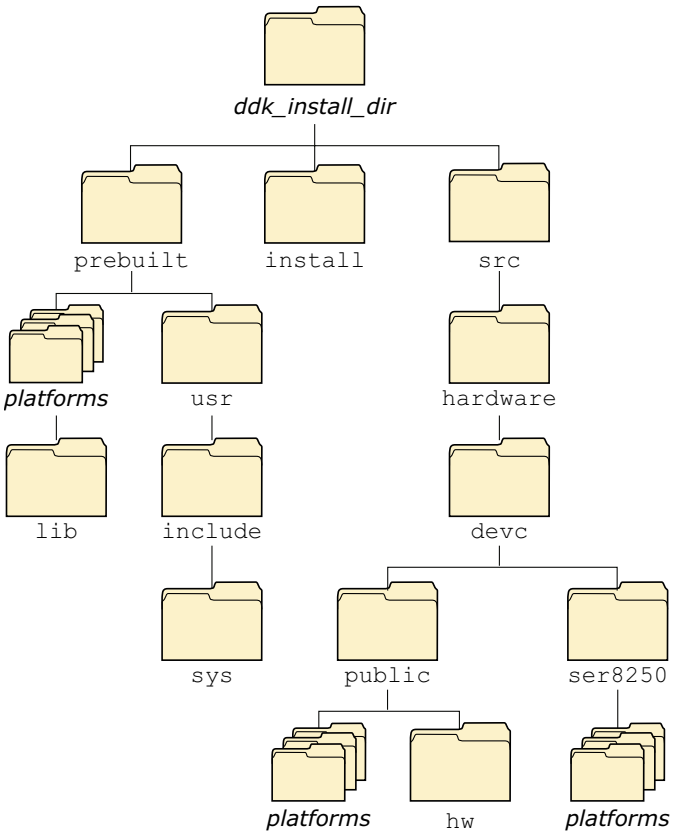
  DDKs are simple zipped archives, with no special requirements. You must manually expand their directory structure from the archive. You can install them into whichever directory you choose, assuming you have write permissions for the chosen directory.

Historically, DDKs were placed in **/usr/src/ddk_VERSION** directory, e.g. **/usr/src/ddk-6.2.1.** This method is no longer required, as each DDK archive is completely self-contained.

The following example indicates how you create a directory and unzip the archive file:

```
# cd ~
# mkdir my_DDK
# cd my_DDK
# unzip /path_to_ddks/ddk-device_type.zip
```

The top-level directory structure for the DDK looks like this:



Directory structure for this DDK.

You must run:

```
. ./setenv.sh
```

before running **make**, or **make install**.

Additionally, on Windows hosts you'll need to run the **Bash** shell (**bash.exe**) before you run the **. ./setenv.sh** command.

If you fail to run the **. ./setenv.sh** shell script prior to building the DDK, you can overwrite existing binaries or libs that are installed in **$QNX_TARGET**.

Each time you start a new shell, run the **. ./setenv.sh** command. The shell needs to be initialized before you can compile the archive.

The script will be located in the same directory where you unzipped the archive file. It must be run in such a way that it modifies the current shell's environment, not a sub-shell environment.

In **ksh** and **bash** shells, All shell scripts are executed in a sub-shell by default. Therefore, it's important that you use the syntax

```
. <script>
```

which will prevent a sub-shell from being used.

Each DDK is rooted in whatever directory you copy it to. If you type **make** within this directory, you'll generate all of the buildable entities within that DDK no matter where you move the directory.

all binaries are placed in a scratch area within the DDK directory that mimics the layout of a target system.

When you build a DDK, everything it needs, aside from standard system headers, is pulled in from within its own directory. Nothing that's built is installed outside of the DDK's directory. The makefiles shipped with the DDKs copy the contents of the **prebuilt** directory into the **install** directory. The binaries are built from the source using include files and link libraries in the **install** directory.

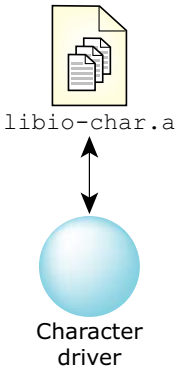
Chapter 1

Character I/O Architecture

In this chapter. . .

Overview

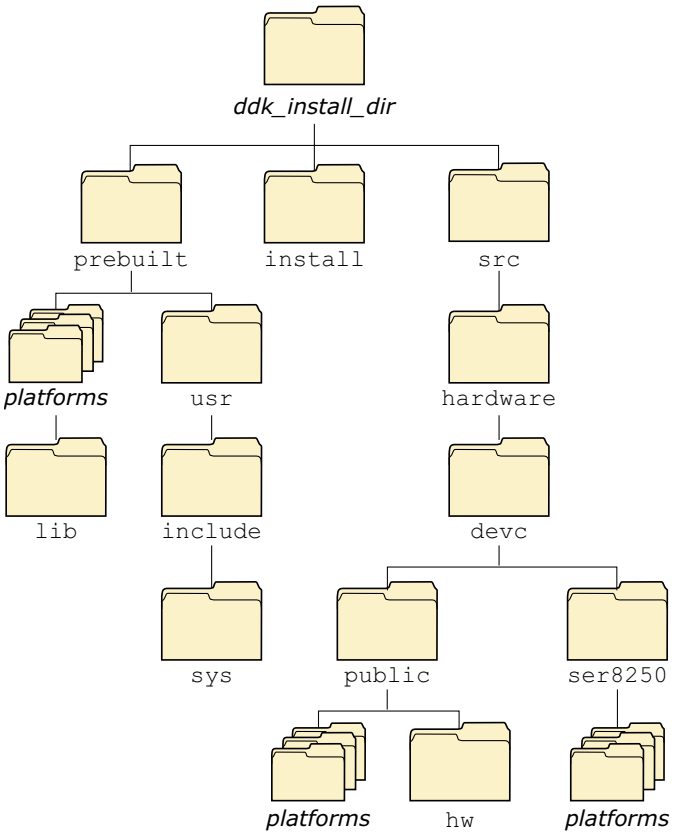
At present, each character driver is a separate process. Each driver links against the **`libio-char.a`** library:



Current Character I/O architecture

DDK source code

When you install the DDK package, the source is put into a directory under the **`ddk_install_dir ddk-char`** directory. Currently, the directory structure for the Character DDK looks like this:



Directory structure for the Character DDK.

Chapter 2

8250 Serial Driver

In this chapter. . .

Creating a serial driver

The Character DDK currently includes the source code for the 8250 serial driver. You may not have to change much:

- If your serial hardware is completely compatible with the 8250, you might not have to change anything.
- If your hardware is almost compatible with the 8250, you might have to change the register addresses. See “Registers,” below.
- If compatability is in question, you may have to change the source code. See “Source code,” below.

Registers

You’ll find the register addresses defined in **ddk_working_dir/ddk-char/src/hardware/devc/public/hw/8250.h**.

The **<8250.h>** file defines:

- the register addresses, specified as offsets from the port address that you set when you start the **devc-ser8250** driver
- bit definitions for the registers.

See the documentation for your hardware for information about its registers and bit definitions.

Source code

The source code for the 8250 serial driver is in **ddk_working_dir/ddk-char/src/hardware/devc/ser8250**. This directory includes:

externs.c	Defines the global data.
externs.h	Includes the required headers and declares the global data.
init.c	Initialization code.

intr.c Interrupt handler routines.

main.c The main part of the driver.

options.c Parses the driver's command-line arguments.

proto.h Prototypes for the driver's interface routines.

query_defdev.c

Queries the default devices. Note that there's a special version of this routine for x86 desktop systems in **x86/query_defdev.c**. For other platforms, there aren't any default devices.

tedit.c The tiny edit-mode routine.

tto.c A routine to transmit a byte, called by **io-char**. It also provides support to control and read hardware control lines status, and provides support for the **stty** utility. **io-char** down call that uses the **stty** command to send output such as line ctrl and line status to the hardware.

There are also platform-specific directories, each of which includes:

<sys_ttyinit.c>

Initialize the tty structure that the driver passes to **io-char**.

Change as little of the given source code as possible, because it's easy to mess things up.

The most important parts of the code are those associated with output and interrupts.

Interrupts

Different chips use interrupts in different ways. Typically, interrupts occurs when:

- A character arrives at the chip. This character is added to the input queue.

  If the device is in edited mode, the character is also added to the canonical queue. Typically, the driver doesn't worry about raw and edited modes; **`io-char`** handles them.
- The chip's transmission buffer is ready for a character.
- A modem-control signal (e.g. hardware flow control) is received.
- An error (e.g. line status, parity error, or framing error) occurs.

Functions

The **`ser8250`** driver includes the following functions, defined in **`proto.h`**:

- *create_device()*
- *options()*
- *query_default_device()*
- *ser_intr()*
- *ser_stty()*
- *sys_ttyinit()*
- *tto()*

The driver's *main()* routine (defined in **`main.c`**) calls:

- *ttc()* with an argument of TTC_INIT_PROC to allocate and configure the resources shared by all devices, e.g. the resource manager.
- *ttc()* with an argument of TTC_INIT_START to allow the driver to start accepting messages, i.e. work.
- *options()* to parse the driver's command-line options.

create_device()

This function is defined in **`init.c`**. The prototype is:

```
void create_device( TTYINIT *dip,
                    unsigned unit )
```

This function gets a device entry and its input/output buffers and creates a new device based on options passed in.

options()

This function is defined in **`options.c`**. The prototype is:

```
unsigned options( int argc,
                  char *argv[] )
```

This function parses the driver's command-line arguments. For information about the arguments, see **`devc-ser8250`** in the *Utilities Reference*.

Depending on the options specified, this function may call:

- *ttc()* with an argument of TTC_INIT_RAW to configure the terminal to RAW mode.
- *sys_ttyinit()* to initialize the **`tty`** as appropriate for the CPU platform.
- *ttc()* with an argument of TTC_SET_OPTION to pass standard terminal configuration options to **`<libio-char.a>`** to be executed.
- *create_device()* to create a device.
- *query_default_device()* to query the default devices if none is specified on the command line.

The *options()* function returns the number of ports.

query_default_device()

This function is defined in **`query_defdev.c`**. The prototype is:

```
void *query_default_device( TTYINIT *dip,
                            void *link )
```

This function returns a placeholder that's used for overwrites in the platform directory.

ser_intr()

This function is defined in **`intr.c`**. The prototype is:

```
const struct sigevent *ser_intr( void *area,
                                 int id )
```

The *ser_attach_intr()* function, which is called by *create_device()*, calls *InterruptAttach()* (see the QNX *Library Reference*) to attach *ser_intr()* to the first handler.

The *ser_intr()* function calls:

- *tti()* to pass a character of data received by the hardware to the **`io-char`** library.
- *tto()* to transmit a character by taking the next available byte in the **`io-char`** lib output buffer and writing it to the hardware.

ser_stty()

This function is defined in **`tto.c`**. The prototype is:

```
void ser_stty( DEV_8250 *dev )
```

This function configures hardware registers and settings such as baud rate, parity, etc.

sys_ttyinit()

This function is defined in **<sys_ttyinit.c>** in the platform-specific directories under **ddk_working_dir/ddk-char/src/hardware/devc/ser8250**.

The prototype is:

```
void sys_ttyinit( TTYINIT *dip )
```

This function initalizes the TTYINIT clock and divisor default as appropriate for the platform.

tto()

This function is defined in **tto.c**. The prototype is:

```
int tto( TTYDEV *ttydev,
         int action,
         int arg1 )
```

This function takes data from **io-char**'s output buffer and gives it to the hardware. It also deals with **stty** commands, by calling *ser_stty()* and provides line ctrl and line status information.

The arguments are:

ttydev A pointer to the driver's **TTYDEV** structure.

action One of:

- TTO_STTY — an **stty** command was received. It's called by **io-char** when the **stty** command is performed on the device. This action calls *ser_stty()*; the argument is ignored.
- TTO_CTRL — set the characteristics of the port i.e. control RS-232 modem lines.
  - *arg1* _SERCTL_BRK_CHG — called by **io-char** when the application requests a break such as *tcsendbreak()* to be sent
  - *arg1* _SERCTL_DTR_CHG — changes the DTR line

- *arg1* _SERCTL_RTS_CHG — used to change the RTS line; **`io-char`** calls this to assert hardware flow control when the input buffer is filling up (based on the highwater level)

- TTO_LINESTATUS — a request for line status. Returns the status of the Modem Status and Modem Control registers when the user performs a *devctl()* with DCMD_CHR_LINESTATUS; the argument is ignored.
- TTO_DATA — output transmit data.
- TTO_EVENT — ignored.

arg1 A data value which has different meanings for different actions. It's used to pass flags that modify the action.

Chapter 3

Character I/O Library

The **libio-char.a** library defines these functions and data types:

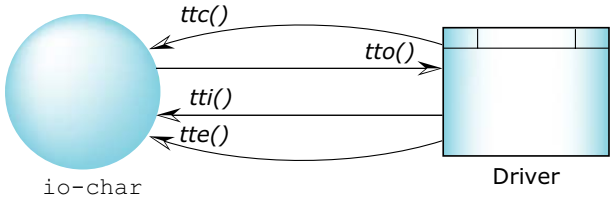
ttc()	Used during initialization to configure the terminal's settings.
tti()	Passes rx data and control information.
tto()	Writes tx data to hardware, handles settings, line control and line status.
TTYCTRL	Contains the settings which are shared by all devices, e.g. the resource manager configuration.
TTYDEV	Contains the settings specific to one serial device.
TTYINIT	Initializes the driver, **termios**, and buffer size.

The **io-char** utility calls the *tto()* function and the driver implements it. The **TTYCTRL** and **TTYDEV** structures provide the interface between **io-char** and the driver. The *tto()* function writes tx data, line status, device settings, and line ctrl information to the hardware.

The driver calls the *ttc()* and *tti()* function calls. The *ttc()* function initializes the device and the resource manager. The *tti()* function passes receive data and control info to the **io-char** utility.

The *tte()* function is generated by an event which causes **io-char**'s event handler to be called.

The relationship between the **io-char** utility and the driver is seen here:



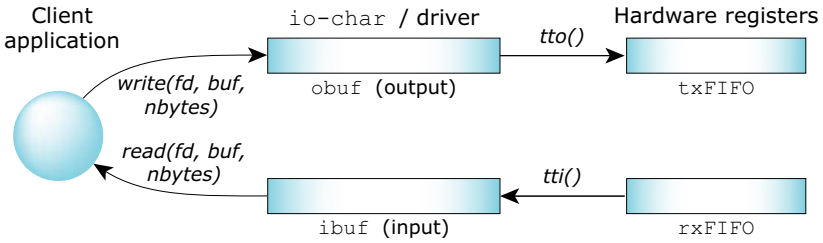
Relationship between **io-char** *and the driver*

The **TTYDEV** structure contains two buffers: an *obuf* (output buffer) and an *ibuf* (input buffer).

The *tto()* function call provides the interface between the Tx FIFO register and the *obuf*. It's called to send the contents of the output buffer to the Tx FIFO register.

The *tti()* function call provides the interface between the Rx FIFO register and the *ibuf*. It's called to place the data from the Rx FIFO register into the input buffer.

The relationship between the output and input buffers and the *tto()* and *tti()* function calls can be seen here:



Buffer and function call interaction

The following table indicates the relationship between the driver and these APIs:

The driver implements:
tto() — to tx data, and perform line status, line ctrl, and device settings, e.g. baud, parity, etc.)
The driver calls:
ttc() — to initialize the device and resource manager
tti() — to pass rx data and control info to **io-char**
The driver generates an event:

continued...

The driver implements:
tte() — to cause the **`io-char`** *tte()* event handler to be called

Controls the terminal

Synopsis:

```
#include <sys/io-char.h >

int ttc(int type,
        void *ptr,
        int arg );
```

Arguments:

type One of:

- TTC_INIT_PROC — allocates and configures the basic resources which are shared by all terminal sessions
- TTC_INIT_CC — configures the character codes for the terminal
- TTC_INIT_RAW — set the terminal into RAW mode
- TTC_INIT_EDIT — set the terminal into EDIT i.e. “cooked” mode
- TTC_SET_OPTION — pass the standard terminal configuration options to **io-char** library for handling.

  If *opt* is found in the common string of options, IO_CHAR_COMMON_OPTIONS, the handler string returns 0. If *opt* is not found, it returns the *opt* back.
- TTC_INIT_START — allow the driver to start accepting messages
- TTC_INIT_TTYNAME — sets up the device name based on the unit number passed in and must be called before TTC_INIT_POWER and TTC_INIT_ATTACH
- TTC_INIT_POWER — initializes power managememt related data structures to defaults (ACTIVE mode only). The driver’s call to TTC_INIT_POWER is mandatoy.

  TTC_INIT_POWER must be called before any calls to **io-char** functions such as *tti()*, or before interrupt handlers are attached.

This *type* must also be called after TTC_INIT_TTYNAME and before TTC_INIT_ATTACH. For power managed device drivers, the *iochar_regdrv_power()* function should be called prior to calling TTC_INIT_POWER.

- TTC_INIT_ATTACH — attaches the resource manager to the name initialized by TTC_INIT_TTYNAME
- TTC_TIMER_QUEUE — register to receive an event once a timer expires
- TTC_INIT_PTY — needed by **`devc-pty`** *only*. Do not use.

ptr A pointer to the structure which will be updated with the new configuration data. Depending on the *type* argument, this argument will be a pointer to a structure of type **`TTYCTRL`**, **`TTYDEV`**, or **`TTYINIT`**.

arg Data which describes the new setting. The values which are valid for this argument vary depending on the *type* argument.

Description:

This function configures the terminal's settings.

Returns:

0 Success.

-1 An error occurred.

Classification:

QNX Neutrino

Safety	
Cancellation point	No
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	No

See also:

tti(), *tto()*, **TTYDEV**

Synopsis:

```
#include <sys/io-char.h >

int tti(TTYDEV *dev,
        unsigned c );
```

Arguments:

dev A pointer to the structure that represents the specific device data has been received on.

c Contains received data and control codes which modify how the data is read and processed. See the TTI_* defines below for more details.

Description:

This function forwards data received by the hardware to **io-char** and passes error/control codes.

The control type is extracted from *c*, and is one of:

TTI_BREAK	Indicates a “break” signal has been detected by the hardware or VINTR character received.
TTI_QUIT	Internal to **io-char**. Indicates a VQUIT character has been received.
TTI_SUSP	Internal to **io-char**. Indicates a VSUSP character has been received.
TTI_OVERRUN	An overrun has been detected by the hardware.
TTI_FRAME	A framing error has been detected by the hardware.
TTI_PARITY	A parity error has been detected by the hardware.
TTI_CARRIER	Indicates to the **io-char** library that a carrier was detected, i.e. the hardware modem is online.

TTI_HANGUP	Indicates to **io-char** that the hardware modem is "hung up." This type is the opposite of TTI_CARRIER
TTI_OHW_STOP	Used by hardware flow control to stop output.
TTI_OHW_CONT	Used by hardware flow control to start output.

Returns:

If this call returns 0, do nothing. If it returns -1 an event needs to be generated for **io-char**.

Classification:

QNX Neutrino

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

See also:

ttc(), *tto()*, **TTYDEV**

TTYCTRL

Tty control structure

Synopsis:

```
typedef struct chario_entry {
dispatch_t *dpp;
int coid;
int timerid;
unsigned max_devs;
unsigned num_devs;
struct sigevent event;
struct sigevent timer;
struct ttydev_entry *timer_list;
unsigned num_events;
struct ttydev_entry **event_queue;
intrspin_t lock;
} TTYCTRL;
```

Description:

A character driver shares the **TTYCTRL** with the **io-char** library. This structure is used to coordinate events, timers, and so on.

The members include:

dpp	A dispatch handle, returned by *dispatch_create()*. Used *only* by **io-char**.
coid	The connection ID. Used *only* by **io-char**.
timerid	The timer ID. Used *only* by **io-char**.
max_devs	Used by **io-char** and the driver to define the maximum number of devices supported.
num_devs	Used *only* by **io-char** to define the current number of devices supported.
event	Used by the driver to send pulse events to **io-char's** event handler. Flags are used to indicate which event occurred. The driver must send the event to **io-char**. The following events are currently defined:

- EVENT_QUEUED — there is an event queued.
- EVENT_SIGBRK — POSIX job control for SIGBRK sends SIGINT. This event is called by TTI_BREAK, so the driver probably doesn't need to do this.
- EVENT_SIGHUP — POSIX job control, TTI_HANGUP.
- EVENT_TTO — not used.
- EVENT_READ — used by **io-char**.
- EVENT_WRITE — called by the driver. Unblock an application waiting to write when the output buffer has room to take characters.
- EVENT_DRAIN — called by the driver. The output buffer has drained (unblock someone waiting on the device to drain.)
- EVENT_TIMEOUT — used by **io-char**.
- EVENT_NOTIFY_INPUT — input notification (used by **io-char**). See the *notify* entry in **TTYDEV**.
- EVENT_NOTIFY_OUTPUT — output notification (used by **io-char**. See the *notify* entry in **TTYDEV**.
- EVENT_NOTIFY_OBAND — driver notifies **io-char** if out-of-band data is available.
- EVENT_CARRIER — generated by TTI_CARRIER.
- EVENT_SIGQUIT — job control, generated by TTI_QUIT to notify that a QUIT character has been received.
- EVENT_SIGSUP — job control, generated by TTI_SUSP to notify that a SUSP character has been received.

timer A pulse to call the timer handler. Used *only* by **io-char**.

timer_list	Used *only* by **io-char**, it provides a list of active timers.
num_events	Used by **io-char** and the driver, it displays the current number of events for processing.
event_queue	An array of **TTYDEV** structures used by **io-char** and the driver to queue events.
lock	A lock used to control access to this structure. Use the *dev_lock()* and *dev_unlock()* macros to access this member.

Classification:

Photon

See also:

TTYDEV

TTYDEV

Structure for a tty device

Synopsis:

```
typedef struct ttydev_entry {
iofunc_attr_t attr;
iofunc_mount_t mount;
TTYWAIT *waiting_read;
TTYWAIT *waiting_write;
TTYWAIT *waiting_drain;
int c_cflag;
int c_iflag;
int c_lflag;
int c_oflag;
volatile unsigned flags;
volatile unsigned       xflags;
    int bcnt;
int fwdcnt;
struct ttydev_entry *timer;
int   timeout;
int   timeout_reset;
union {
int tmrs;
struct {
char spare_tmr;
char tx_tmr;
char brk_tmr;
char dtr_tmr;
} s;
}  un;
pid_t brkpgrp;
pid_t   huppid;
    cc_t   c_cc[NCCS];
unsigned char   fifo;
unsigned char   fwd;
unsigned char   prefix_cnt;
unsigned char   oband_data;
int   highwater;
int   baud;
struct winsize winsize;
TTYBUF   obuf;
TTYBUF   ibuf;
TTYBUF   cbuf;
iofunc_notify_t   notify[3];
struct ttydev_entry *extra;
```

```
TTYWAIT *waiting_open;
void *reserved2; /* reserved for use by io-char */
int (*io_devctlext)(resmgr_context_t *ctp, io_devctl_t *msg, iofunc_o
char   name[TTY_NAME_MAX];
} TTYDEV;
```

Description:

A character driver shares the **TTYDEV** structure with the **io-char** library.

This structure is used to handle devices shared between the driver and **io-char**.

The members include:

attr	A resource manager attribute
mount	Related to resource manager information
waiting_read	The queue to store blocking clients waiting to read
waiting_write	The queue to store blocking clients waiting to write
waiting_drain	The queue to store blocking clients waiting to drain.
c_cflag	POSIX **termios** flag describing the hardware control of the terminal
c_iflag	POSIX **termios** flag describing the basic terminal input control
c_lflag	POSIX **termios** flag used to control various terminal functions
c_oflag	POSIX **termios** flag describing the basic terminal output control
flags	The following flags are currently defined:

- OHW_PAGED — the output hardware flow control (set by **io-char** and used by the driver)
- IHW_PAGED — input hardware flow control is asserted; the device's highwater mark has been reached and doen't want to receive any more data. This flag also asserts the RTS line.
- OSW_PAGED — output software flow control is asserted; the device should not transmit any data (set by **io-char** and used by the driver)
- ISW_PAGED — input software flow control is asserted; the device's highwater mark has been reached and doesn't want to receive any more data. This flag also transmits VSTOP.
- EDIT_INSERT — for edit mode. Insert or overstrike typing mode.
- EDIT_PREFIX — for edit mode. Look for edit keys which begin with a fixed prefix, e.g. ESC [ansi**"** used with POSIX c_cc[VPREFIX].
- OBAND_DATA — indicates that out-of-band data is available
- LOSES_TX_INTR — set if the hardware loses the tx interrupt. Causes a periodic timer to call *tto()* to transmit data.
- TIMER_ACTIVE — used by **io-char**
- TIMER_KEEP — used by **io-char**
- NOTTY — used by PTYs
- NL_INSERT — used to notify application if a **\n** was changed to a **\r**
- ISAPTY — used by PTYs
- PTY_MASTER_ONLY — used by PTYs
- LITERAL — used by **io-char**
- FIRST_TIME_ALONE — used by **io-char**

xflags	OSW_PAGED_OVERRIDE — override OSW_PAGED to allow transmission of controlled characters when in a software flow control suspend state. This flag is set by **io-char** and is used and cleared by the driver.
bcnt	Internal to **io-char** and used to determine the number of bytes needed to notify a read client.
fwdcnt	Internal to **io-char** and used to determine the number of fwd counts.
timer	Used by **io-char**.
timeout	Used by **io-char**.
timeout_reset	Used by **io-char**.
tmrs	One of several available for **io-char** to use.
spare_tmr	Spare used only by **io-char** for drain.
tx_tmr	Enabled by LOSES_TX_INTR. The timer causes *tto()* to be called to work around some parts that lose transmit interrupts.
brk_tmr	Used only by **io-char** sending break; calls *tto()* (TTO_CTRL, *dtrchg*).
dtr_tmr	Used by **io-char** to set *dtr* line i.e. generate SIGHUP calls *tto()* (TTO_CTRL, *dtrchg*).
brkpgrp	Used by **io-char**.
huppid	Used by **io-char**.
c_cc	POSIX special control-characters.
fifo	Used only by the driver.
fwd	Forward character used by **io-char**. It's used with *fwdcnt* to implement *forward*, described in *readcond()*.

prefix_cnt	For **io-char** only.
oband_data	Out-of-band data set by the driver in **<intr.c>**. The application gets it from **io-char** via a *devctl()*.
highwater	Set by the driver and used by **io-char** to determine when to invoke flow control. (Make sure this value is *LESS* than the input buffer size).
baud	The device's baud rate.
winsize	Used only by **io-char**.
obuf	The output buffer.
ibuf	The input buffer.
cbuf	The canonical buffer.
notify	The notify list. It implements *iofunc_notify_trigger()* resource manager information. The following arguments are used: • *notify[0]* — notify for input used by **io-char** • *notify[1]* — notify for output to the driver, **<tto.c>** • *notify[2]* — notify for data that out-of=band to the driver, **<intr.c>**
extra	Used for PTYs.
waiting_open	The queue to store blocking clients waiting to open.
io_devctlext	Custom **devctl** command.
name	The device's name i.e. **/dev/ser1**

Classification:

QNX Neutrino

See also:

TTYCTRL

TTYINIT

Structure for a tty device

Synopsis:

```
typedef struct ttyinit_entry {
    _Paddr64t               port;
    unsigned                port_shift;
    unsigned                intr;
    int                     baud;
    int                     isize;
    int                     osize;
    int                     csize;
    int                     c_cflag;
    int                     c_iflag;
    int                     c_lflag;
    int                     c_oflag;
    int                     fifo;
    int                     clk;
    int                     div;
    char                    name[TTY_NAME_MAX];
 {  TTYINIT;
```

Description:

A character driver shares the **TTYINIT** with the **io=char** library. This structure is used to initialize baud rate, input, output, canonical buffer sizes, **termios** flags, interrupts, etc.

The members include:

port — Contains addresses of device registers.

port_shift — Used to provide spacing between registers. For example:

- 0 — is for 8-bit registers
- 1 — is for 16-bit registers
- 2 — is for 32-bit registers

intr — The interrupt number associated with the device.

baud — The device's baud rate.

isize	The input buffer size.
osize	The output buffer size.
csize	The canonical buffer size.
c_cflag	See **TTYDEV**.
c_iflag	See **TTYDEV**.
c_lflag	See **TTYDEV**.
c_oflag	See **TTYDEV**.
fifo	See **TTYDEV**.
clk	The clock frequency is used with baud rate and divisor in **stty**.
div	The divisor is used with baud rate and clock in **stty**.
name	The name of the device.

Classification:

QNX Neutrino

See also:

TTYDEV

Index