QNX® Neutrino® Realtime Operating System

QNX® 4 to QNX® Neutrino®
Migration Guide

For QNX® Neutrino® 6.2

Published under license by:

QNX Software Systems International Corporation
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 2006

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 web-based **Developer Support Center**.

Contents

About This Guide

The *QNX 4 → QNX Neutrino Migration Guide* is intended as a kind of road map to help you:

- discover the differences and additional functionality of QNX Neutrino
- modify your existing QNX 4 source code to work under QNX Neutrino.

Written specifically for the QNX 4 applications developer, the guide's main focus is the API; it doesn't deal with differences in system commands, editors, and so on. Nevertheless, we hope you'll find the guide a valuable resource for determining what you need to do to your existing software in order to take advantage of the rich new features of QNX Neutrino.

The following may help you find what you need in this guide:

When you want to:	**Go to:**
Get an overview of the differences between the two systems	Meet the New OS
Plan your migration strategy	Migration Methodology
Find out about headers, libraries, debugging, buildfiles, and more	Development Environment
Learn about threads, connection IDs, pulses, resource managers, and more	Programming Issues
Look up a cover function	The Migration Library
Identify items in your QNX 4 programs that need attention	The **`mig4nto`** Utility
Look up the QNX Neutrino equivalent for any QNX 4 function	QNX 4 Functions & QNX Neutrino Equivalents
View the list of QNX 4 functions supported by QNX Neutrino	QNX 4 functions supported by QNX Neutrino

Chapter 1

Meet the New OS

In this chapter. . .

Architecture

Architecturally, QNX Neutrino is the same as QNX 4. It provides an open-systems POSIX API in a robust scalable form suitable for a wide range of solutions — from tiny resource-constrained systems to high-end distributed computing environments.

Support for multiple processors

One big difference between QNX 4 and QNX Neutrino is that the new OS runs on a number of different processors — x86, ARM, MIPS, PowerPC, and SH4. Assuming that you didn't use any processor-specific tricks, a program written for QNX Neutrino on an x86 could be recompiled and relinked so that it will also run on a Power PC machine.

SMP

QNX Neutrino can also run on a multi-processor machine. This machine might have 4 Gbytes of physical memory managed by a number of processors.

Portability

QNX Neutrino was also designed to be even more portable than QNX 4. The new OS contains newer POSIX components (e.g. POSIX threads). New functions from the Unix 98 standard and from other sources have been added.

Moving files from QNX 4 to QNX Neutrino

Migrating files from a QNX 4 system to a QNX Neutrino system isn't complicated for this simple reason: they use the same filesystem. If you're going to run QNX Neutrino on a PC with a QNX 4 filesystem, then all the files in the QNX 4 partition or disk are accessed exactly the same as they were before.

In fact, you could replace your **`/.boot`** or **`/.altboot`** file with a QNX Neutrino bootable image and reboot into the new QNX Neutrino OS.

Typical development cycle

Many developers may want to have their QNX Neutrino target machine act as an NFS or CIFS client; this is certainly possible.

Using this scheme, you can set up a QNX 4 development system and a QNX Neutrino target system that has access to the QNX 4 files. With two machines side-by-side, a typical development cycle becomes:

From the QNX 4 machine:

1. Edit.
2. Compile.
3. Link.

From the QNX Neutrino machine:

➤ Run.

Chapter 2

Development Environment

In this chapter. . .

Compiler & tools

The compiler used for QNX Neutrino is the GNU compiler (**gcc**). Currently, development can be done from these hosts:

- QNX Neutrino
- MS-Windows
- Solaris

If you have the QNX Momentics Professional Edition, you can develop using the Integrated Development Environment (IDE) from any host. Alternatively, you can use command-line tools that are based on the GNU compiler.

If you have the QNX Momentics Standard Edition, you have only the command-line tools available.

For MS-Windows hosts, you also have the option of getting the CodeWarrior tools from Metrowerks. Currently, the CodeWarrior IDE also uses **gcc**.

Note that when developing using the GNU compiler, you don't run the compiler directly. Instead, you use a front-end tool called **qcc**. A minimum compile would look like this:

```
qcc myprogram.c -o myprogram
```

Header files

Header files reside under **${QNX_TARGET}/usr/include**, where the value of **QNX_TARGET** depends on where you're doing your development.

Libraries

Libraries that you link against are in **${QNX_TARGET}/${PROCESSOR}/lib**.

When migrating from QNX 4, *PROCESSOR* would most likely be x86.

Static and dynamic libraries

QNX Neutrino supports both static libraries and dynamic libraries. If you link with static libraries, then code from the libraries is inserted into your executable.

Dynamic libraries in QNX Neutrino are the equivalent of shared libraries in QNX 4. In fact, we usually call them *shared objects*, though many people also know them as DLLs. In the case of dynamic libraries, the code for the library is loaded into memory when the first program that uses that library is run.

Useful manifests

Here are some manifests you may find useful:

```
#if defined(__WATCOMC__)
    /* Then the program was compiled with Watcom */
#elif defined(__GNUC__)
    /* Then the program was compiled with GCC */
#elif defined(__MWERKS__)
    /* Then the program was compiled with Metrowerks */
#endif
```

As well, you could do things like:

```
#if defined(__QNXNTO__)
    /* Then the program was compiled for QNX Neutrino */
#else
    /* Then the program was compiler for QNX 4 */
#endif
```

Debugging

There are a variety of options that you can use for debugging:

- the IDE that's part of the QNX Momentics Professional Edition
- **gdb**, which is available on all hosts
- the CodeWarrior IDE debugger from Metrowerks
- **ddd** under QNX Neutrino.

The **gdb** debugger is a command-line program used in conjunction with the **gcc** compiler (recall that **gcc** is the back-end compiler for the **qcc** command) and is documented in the *Utilities Reference*.

Buildfiles and images

Image files are conceptually the same as in QNX 4, but structurally very different. There is a **sysinit** file if you're using QNX Neutrino, but there isn't one by default on your target. The buildfile language has been expanded to include primitive scripting.

For more information, see the chapter on "Making an OS Image" in the *Building Embedded Systems* book in the Embedding SDK package, as well as the documentation on the **mkifs** utility.

Chapter 3

Programming Issues

In this chapter...

Scheduling

The main difference in scheduling between QNX 4 and QNX Neutrino is that scheduling is done *by thread*, not by process. In QNX Neutrino, the highest-priority thread is chosen to run, regardless of what process it's in.

This has some interesting ramifications. For instance, from a QNX 4 perspective, a process can preempt itself! Of course, this is minimized when migrating to QNX Neutrino, since all your processes will be single-threaded.

Priority range

QNX Neutrino extends the priority range (from 0 to 31) to **0 through 63**. Higher numbers still represent higher priority. The *sched_get_priority_min()* and *sched_get_priority_max()* calls simply return the minimum and maximum priority; the special *idle* thread (in the process manager) has priority 0.

At each priority, the threads in QNX Neutrino are scheduled as in QNX 4, with the exception that there's no longer an adaptive scheduling policy. The available policies are FIFO and round-robin, both of which operate the same as in QNX 4.

Remember that these policies come into play only when there's more than one thread ready to run *at the same priority*.

☞ According to POSIX, there's a third scheduling algorithm called SCHED_OTHER, which is up to the OS vendor to decide what it actually means. Currently in QNX Neutrino, SCHED_OTHER is the same as SCHED_RR (round-robin), but that may change some day, so we don't recommend using SCHED_OTHER.

QNX Neutrino supports the *getprio()* and *setprio()* function calls from QNX 4. But because the scheduling in QNX Neutrino is by thread, not by process, there's a caveat here: When attempting to set the priority of a process by calling *setprio()*, thread number 1 in the process has its priority changed, *not* all threads in the process.

If the process ID given to *setprio()* is zero, indicating the current process, it's the *calling thread* within that process whose priority will be set. Since QNX 4 code ported to QNX Neutrino would likely contain contain one thread anyway, this is just what you'd want to have happen.

Because of its increased number of synchronization primitives, as well as the inclusion of threads, QNX Neutrino has more states than QNX 4:

- If a QNX Neutrino thread is in the RUNNING state, then it's the thread that's actually using the CPU. And since QNX Neutrino supports SMP, there could be multiple threads in this state.
- When a thread is in the READY state, it wants to execute on a CPU, but it's not the thread that's executing. This distinction between RUNNING and READY was not visible to the user in QNX 4.
- The SEND-blocked, RECEIVE-blocked, REPLY-blocked, STOPPED, SEM-blocked, and DEAD states are the same as in QNX 4.

QNX Neutrino also has these additional states, which were either slightly different or not present at all in QNX 4:

- If a thread calls the *sigsuspend()* function, it will be SIGSUSPEND-blocked awaiting a signal.
- If the thread calls *sigwaitinfo()*, it's also waiting for a signal to occur; this is the SIGWAITINFO-state.
- A thread can also request to be suspended for a short period of time by calling *nanosleep()*, which puts the thread into the NANOSLEEP state until the time has expired.
- QNX Neutrino adds two synchronization methods (mutex and condvar). If a thread is awaiting one of these, it would be in the MUTEX state or the CONDVAR state until the conditions allow the thread to continue.
- A thread can call *pthread_join()* to await the termination of a child thread. If that thread hasn't terminated yet, the caller is in the JOIN state until the child thread terminates.
- If a thread, such as one in a device driver, is waiting for an interrupt, it could be in the INTR state (*InterruptWait()*) until the interrupt happens and SIGEV_INTR event is delivered to it.

The QNX Neutrino **`pidin`** utility lets you see the thread state under the "STATE" column. It's roughly analogous to **`ps -ef`** in QNX 4. In cases where a thread is blocked awaiting some other thread (e.g. waiting for the reply to a write on a serial port), the "Blocked" column shown by **`pidin`** indicates the thread ID that the thread is blocked on.

Process issues

Process creation

The *qnx_spawn()* function and the *qnx_spawn_options* structure are no longer supported. The *spawn*()* family (*spawnl()*, *spawnve()*, ...) of functions still exist. There's also a new function called *spawn()* that provides much of the functionality in the *qnx_spawn_options* structure.

☞ POSIX has a new function called *posix_spawn()*, which we don't support as of the time of this writing.

Something similar to the io vector (*iov* parameter to *qnx_spawn()* and *iov* member of *qnx_spawn_options*) is available via the *fd_map* to *spawn()*. However, the FDs passed in this will be the only ones open for the child.

When calling the *spawn()* function, note there are some undocumented SPAWN_ flags. These are undocumented, because they're mainly intended for people migrating from QNX 4. They can be found in the **`<spawn.h>`** header file. Some readily recognizable ones are:

- SPAWN_NOZOMBIE
- SPAWN_NEWPGROUP
- SPAWN_HOLD
- SPAWN_SETSID
- SPAWN_TCSETPGROUP

QNX Neutrino also supports *fork()*, but there's the restriction that *fork()* can't be used within a multithreaded process. For this case, you might consider *vfork()* instead.

Process flags

The following are the issues involved with the most frequently used flags you can set in QNX 4 using the *qnx_pflags()* function:

- _PPF_IMMORTAL — This is no longer supported. You can't catch or ignore SIGKILL.
- _PPF_INFORM — This is no longer supported. There's no replacement that will let you know of any process dying. You could periodically poll the process manager by walking through the pids in **`/proc`**, but this is highly inefficient. For other methods, see the section on "Process termination" in the chapter on processes in the *Programmer's Guide*.
- _PPF_PRIORITY_FLOAT — This is the default in QNX Neutrino. To disable this behavior, pass the _NTO_CHF_FIXED_PRIORITY to your call to *ChannelCreate()*.
- _PPF_PRIORITY_REC — This is the behavior in QNX Neutrino. It can't be turned off.
- _PPF_SIGCATCH — To get this behavior, pass the _NTO_CHF_UNBLOCK flag to your call to *ChannelCreate()*.
- _PPF_SERVER — This is no longer supported.

Native QNX networking

The equivalent of FLEET, the QNX 4 native networking, is Qnet. From the command line, instead of using node IDs ("nids", you use node names. From code, instead of using nids, you use node descriptors. For detailed information on these issues, see the sections called Qnet Networking in the *Programmer's Guide* and *System Architecture*.

I/O Managers vs Resource Managers

QNX Neutrino has the same concept as QNX 4 I/O managers, but they're called resource managers instead. In QNX 4, unless you had used the I/O manager framework available as free software, then you'll have to rewrite most of your I/O manager from scratch.

In QNX Neutrino, a resource manager library is provided for you as part of the regular libraries. This library hides a lot of the gory details, allowing you to concentrate on code that's specific to your application while still presenting a POSIX front end to the client.

The downside to this is that migrating I/O managers will likely be the one set of code that involves the most work, because this is where there are the most differences. The work is the least if you had used the I/O manager framework that's available for QNX 4 as free software, since your process will be similar architecturally.

There's a chapter in the the *Programmer's Guide* called "Writing a Resource Manager" that goes into detail on how to write these.

Messages

Connection-oriented philosophy

The QNX Neutrino OS still uses the send/receive/reply model. A receiver still blocks on some receive function call, a sender still sends via some send function call and blocks until the receiver replies.

The functions involved are:

- *MsgReceive*()*
- *MsgSend*()*
- *MsgReply*()*
- *MsgRead*()*
- *MsgWrite*()*

There's still the multipart message option — the above functions whose names contain a "v" suffix are the multipart message versions (e.g. *MsgSendv()* is analogous to *Sendmx()*).

There's also an additional function for replying called *MsgError()*. It takes an errno value as a parameter and causes the *MsgSend*()* to return with -1 and *errno* set to this value. The *MsgReply*()* also has an interesting new status parameter. Whatever you pass for this will be what the *MsgSend*()* returns.

Channel IDs vs process IDs

There are also some fundamental differences. Under QNX 4, the sender sent to a process via a process ID, but this no longer works when there could be multiple threads within the receiving process.

Under QNX Neutrino, some thread in the receiving process creates a channel (via *ChannelCreate()*). Then whichever thread or threads want to receive messages from that channel call a *MsgReceive*()* function, passing it the channel ID (*chid*). So, in QNX Neutrino, you receive using channel IDs, not process IDs.

Some thread in the sending process then creates a connection to that channel (usually via *ConnectAttach()*). Then whichever thread or threads want to send a message will send using the connection ID (*coid*) via a *MsgSend*()* function.

So, in QNX Neutrino, you send through a *connection*, not to a process ID. The *MsgReceive*()* function returns a receive ID (*rcvid*) and passes this to the *MsgReply*()*. So in QNX Neutrino, you reply to a receive ID, not a process ID.

Notice that this is connection-oriented, unlike in QNX 4 where any sender could send to any receiver just by passing the receiver's process ID to the send function call. In QNX Neutrino, the receiver must deliberately advertise its channel ID before any sender can create a connection to it and send.

How should the receiver be written?

There are various ways to write the receiver. You have the same options as under QNX 4 — from plain receiver all the way to resource managers (called I/O managers in QNX 4). The difference is we recommend that you write receivers as resource managers in QNX Neutrino applications.

One of the reasons for this recommendation is that the resource manager library takes care of many of the details for you. This is even more important in QNX Neutrino as there are now more details. For example, in QNX 4 when receiving a message from a sender on another node of the native QNX network, the number of bytes received was the *smaller* of what the sender was sending and what the receiver was asking to receive, but no smaller.

Under QNX Neutrino, however, the receiver could potentially have received *less* than what the sender was sending and the receiver was receiving, depending on the packet size of the protocol used. The resource manager library handles this detail for you (if you're not using the resource manager library, see the section "Receiving from across the network" in this chapter.

Although we recommend writing resource managers, we also recognize that people migrating from QNX 4 who used a simple *Receive()* loop may not want to make the many changes required to convert to resource managers. As such, this section will go into some detail on doing proper *MsgReceive()* handling.

How does the sender find the receiver?

We said above that the server creates a channel and advertises the channel ID. The sender then somehow determines that channel ID and connects to it. But how does the server advertise the channel ID?

1 If you're willing to rewrite your receivers, or if you previously wrote them as I/O managers, then you could write receivers as resource managers. In this case, the *ChannelCreate()* is done by the resource manager library and the *ConnectAttach()* is done by the *open()*. The sender finds the receiver by calling *open()*:

```
fd = open(_name_receive_registered, ...);
...
```

```
MsgSend(fd, ...);
```

One thing that falls very nicely from this is that to connect to a server on another node of the native QNX network, you need only put node information on the front of the name you pass to *open()* (e.g. **`fd = open("/net/console/dev/robotarm", ...)`**).

Note that a resource manager is the equivalent of an I/O manager in QNX 4. If you wrote your I/O managers using the iomanager framework in **`/usr/free`** (or had a similar one of your own) then, although you'll have to use different function calls, architecturally the resource manager library and the I/O manager framework are very similar.

2 In QNX Neutrino, there's a set of functions, including *name_attach()* and *name_open()*, that do the job that *qnx_name_attach()* and *qnx_name_locate()* do in QNX 4. Note that global names are supported via the **`gns`** process.

3 If the receiver is parent and the sender is child, then the channel ID can be passed in spawn arguments lists.

4 If you have a starter process that starts the above two processes, then starter could create a channel and pass the channel ID to its children via command-line args. The children would then send their respective channel IDs to starter (effectively registering with starter) and request each other's channel IDs from starter.

The method that you choose depends on how much migrating you wish to do. If you want to do as little as possible, then the migration library is the starting place. If you don't mind rewriting a little of your Send code, then *name_attach()*/*name_open()* might be the way to go. If you're already using I/O managers or want to migrate to resource managers, then go ahead and do so.

☞ This solution is deprecated for Neutrino, version 6.3.0.

Receiving messages in a resource manager

When looking at writing a resource manager, one of the first things you'll wonder about is how to send messages to it and get replies back. There are various ways:

1 The POSIX way is for the sender to use the POSIX *devctl()* function call to send the message. The resource manager would have an **`io_devctl`** handler registered for processing the message. The only disadvantage to this approach is that there's only one message buffer parameter and one size parameter in the *devctl()* call. This means that if you want to send a 10-byte message and get a 1000-byte reply back, you must provide a 1000-byte buffer and specify a size of 1000 bytes. Even though the buffer contains only 10 bytes of data for the send message, it will send the entire 1000 byte buffer.

2 Another way is to use *message_attach()* to register a range of message types and a handler to be called whenever a message is received whose type is in that

range. With this method, the sending can be done via *MsgSend*()* and the entire contents of the send message and the reply message are in your control.

3. You can send messages using *MsgSend*()* by putting a header of type **`io_msg_t`** at the front of your message. Set the type member to _IO_MSG. The resource manager would register a msg handler in the **`resmgr_context_funcs_t`** structure, and when a message of type _IO_MSG arrives, your msg handler will be called with the message. The reply can be anything.

Don't forget that the above steps show how to send a message and get a reply back. If all your client wants to do is send some data, then the *write()* function call may be all you need. The *read()* function can be used for the opposite direction.

_PPF_SIGCATCH

Just as QNX 4 has the _PPF_SIGCATCH flag, QNX Neutrino has the _NTO_CHF_UNBLOCK flag for the same reasons. In QNX Neutrino, the flag is set for the channel that's being received on — it's passed to *ChannelCreate()*. One difference is that in QNX 4 this affected REPLY-blocked senders who are hit with a signal. In QNX Neutrino, it still affects REPLY-blocked senders who are hit with a signal, but also if they want to time out (via *TimerTimeout()* or *timer_timeout()*).

The _NTO_CHF_UNBLOCK flag is automatically set for resource managers. If the receiver is a resource manager, then when the REPLY-blocked sender wants to unblock, the resource manager library will call an io_unblock handler. If you don't provide an io_unblock handler, then default handling will be done for you resulting in your client potentially not unblocking when it wants to. The *rcvid* member of the **`resmgr_context_t structure`** (**`ctp->rcvid`** — you'll learn about this structure when you do resource managers) is the one you would reply to and/or use to look up the sender in a list of blocked senders.

If you're calling *MsgReceive*()* directly, then a pulse will arrive from the kernel (pulses are discussed later). The code member of the pulse message will be _PULSE_CODE_UNBLOCK and the value member of the pulse message will be the receive ID from the time that the *MsgReceive*()* received the sender's message. This receive id is who you would reply to and/or use to look up the sender in a list of blocked senders. Note that it's generally not a good idea to keep the receive id around once the reply has been done, because after that point its value is recycled.

Message priority

In QNX 4, the _PPF_PRIORITY_REC flag would be used to have messages be received in the order in which they were sent. In QNX Neutrino, this is automatically the behavior of the receiver.

Priority floating

The behavior that you get in QNX 4 by setting _PPF_PRIORITY_FLOAT is now the default in QNX Neutrino. To have your receiver's priority not float, set the _NTO_CHF_FIXED_PRIORITY flag in the call to *ChannelCreate()*.

Receiving from across the network

As mentioned above, when receiving a message from a sender on another node of the native QNX network, it's possible that the number of bytes received could be smaller than both what the sender was sending and what the receiver was asking for. The fourth parameter passed to your *MsgReceive*()* is an info parameter of type **`struct _msg_info`**. It has a member called *msglen*, which contains the number of bytes that were actually copied into your receive buffer. It also has a member called *srcmsglen* which will contain the number of bytes that the sender wants to send, but only if you pass the _NTO_CHF_SENDER_LEN flag when calling *ChannelCreate()*. So a code snippet for handling this situation would be:

```
int              chid, rcvid;
struct _msg_info info;
my_msg_t         msg;

chid = ChannelCreate (_NTO_CHF_SENDER_LEN);

for (;;) {
    rcvid = MsgReceive (chid, &msg, sizeof(msg), &info);
    if (rcvid > 0 && info.srcmsglen > info.msglen &&
            info.msglen < sizeof(msg)) { // got it all?
        int nbytes;

        if ((nbytes = MsgRead_r(rcvid, (char *) msg + info.msglen,
                sizeof(msg) - info.msglen, info.msglen)) < 0) {
            MsgError(rcvid, -nbytes); // nbytes contains an errno value
            ...
        }
        ...
    }
    // now we have it all
    ...
}
```

Events

Later in this chapter we'll look at pulses, a replacement for QNX 4 proxies that also let you pass a little information along. We'll also examine POSIX signals. Because these (and other) primitives are similar, there's an underlying mechanism called an *event* that handles them.

An event in QNX Neutrino is a form of notification that can come from a variety of places: timer, interrupt handler, your threads, etc. An event can contain a pulse. A user hitting Ctrl – C on a keyboard causes an event containing a signal to be delivered. A timer could expire, delivering an event containing a pulse.

A thread delivers an event to another thread by calling *MsgDeliverEvent()*. We'll see an example of this when we talk about pulses. This function takes a receive id and an event structure of type **`struct sigevent`**. The latter contains several fields, including:

sigev_notify	The type of the event, whether it's a signal, a pulse, or whatever.
sigev_priority	The priority of the event; higher numbers mean higher priority.
sigev_code and *sigev_value*	The code and value fields for a pulse.
sigev_signo	The signal number for a signal.

There are macros in the include file **`<sys/siginfo.h>`** that make it simple for you to fill in the fields in this structure.

Proxies vs pulses

QNX 4 proxies have disappeared from QNX Neutrino. They've been replaced by *pulses*. A pulse is like a QNX 4 proxy in one critical way — it's *asynchronous*, so the sending thread doesn't block. But the data for QNX 4 proxies was "canned data" that couldn't be changed from one *Trigger()* call to the next. With pulses, the data can be different from one "trigger" to the next.

Each QNX Neutrino pulse carries with it two items of information:

- an 8-bit value known as the "code"
- a 32-bit value called the "value"

Although the "code" is a signed quantity, you should use only values in the range _PULSE_CODE_MINAVAIL to _PULSE_CODE_MAXAVAIL. The remaining code values are reserved for the OS.

Pulses are received by having your resource manager register a pulse-handler function. This is done by calling *pulse_attach()*. When the pulse arrives, the handler will be called.

If you're not writing a resource manager, then pulses can also be received by the *MsgReceive*()* and *MsgReceivePulse*()* functions. The return value will be zero and the message buffer will contain a message of type **`struct _pulse`**.

In QNX 4, at setup time, the receiver would typically attach a proxy and send the proxy id to the process doing the triggering. Whenever necessary, the triggerer would trigger the proxy. In QNX Neutrino, you'd do something very similar. At setup time, the receiver would fill an event structure with a pulse and send it to the process doing the delivering. Whenever necessary, the deliverer would deliver the event using *MsgDeliverEvent()*. As a result, the receiver would receive a pulse message.

There's another function for sending a pulse called *MsgSendPulse()*. When migrating from QNX 4, you would use this to replace *Trigger()* in cases where the triggering process had attached the proxy to the receiver (instead of the receiver attaching the proxy to itself). With *MsgSendPulse()* there's no event structure to fill in.

Example of pulses with a resource manager

The following code snippets illustrate sending and receiving pulses using resource managers. The deliverer is a resource manager called **pulsesnd** — it registers the name **/dev/pulsesnd**.

First of all, since we have two processes communicating with each other, we have the following in a common header file:

```
// _IOMGR_PULSESND identifies the pulsesnd resource manager
#define _IOMGR_PULSESND _IOMGR_PRIVATE_BASE

#define PULSESND_SUBTYPE_GIVE_EVENT 1

typedef struct {
    struct _io_msg  hdr;   // standard header for _IO_MSG messages
    struct sigevent event; // the event to deliver
} pulsesnd_io_msg_give_event_t;

// the reply for IO_MSG_SUBTYPE_GIVE_EVENT is empty
```

Next, we have the receiver process that will be receiving the pulse. The following is the code that registers the pulse handler, fills in an event structure, and sends the event structure to the deliverer:

```
main
{
    pulsesnd_io_msg_give_event_t msg;

    ... // setup code for the resource manager goes here

    // register our pulse handler, note that this call will figure out
    // a pulse code for us, pulse_handler() will be called whenever
    // the pulse arrives

    our_pulse_code = pulse_attach (dpp, MSG_FLAG_ALLOC_PULSE,
                                   0, pulse_handler, NULL);

    // send a pulse event structure to pulsesnd, pulsesnd is the
    // process (another resource manager) that will deliver this pulse
    // event when data is available.  When it does, pulse_handler()
    // will be called.

    fd = open ("/dev/pulsesnd", O_RDONLY); // find pulsesnd

    // create a connection to the channel that our resource manager is
    // receiving on

    coid = message_connect (dpp, MSG_FLAG_SIDE_CHANNEL);

    // fill message buffer with an _IO_MSG type message (for this
    // example)
    msg.hdr.type = _IO_MSG;
    msg.hdr.combine_len = sizeof(msg.hdr);
    msg.hdr.mgrid = _IOMGR_PULSESND; /* resmgr identifier */
    msg.hdr.subtype = PULSESND_SUBTYPE_GIVE_EVENT;

    // this macro fills in the event structure portion of the message

    SIGEV_PULSE_INIT(&msg.event, coid, getprio (0), our_pulse_code, 0);

    // send it to pulsesnd so that it can deliver it when it wants to

    MsgSend (fd, &msg, sizeof(msg), NULL, 0);

    ...
}

//
// pulse_handler - Will be called when the pulse is delivered
//
int
pulse_handler (message_context_t *ctp, int code, unsigned flags,
               void *handle)
{
    if ( code == our_pulse_code ) {
        // we got a pulse (we're not expecting any others, this check
        // is for example only)

}
return code;
}
```

Next we have the code where the deliverer receives the event structure. Note that it is really a msg type handler that the resource manager library calls whenever a message of type _IO_MSG arrives. The _IO_MSG that arrives is the message sent to us in the snippet above. This msg type handler is registered with the resource manager library just as you would register a read or write handler.

```
int
io_msg (resmgr_context_t *ctp, io_msg_t *msg, RESMGR_OCB_T *ocb)
{
    pulsesnd_io_msg_give_event_t pmsg;

    // go get the message again to make sure we got it all

    MsgRead(ctp->rcvid, &pmsg, sizeof(pmsg), 0);

    // we need to store away the event and the rcvid if we are to
    // deliver the pulse later.  This is the same idea as saving away
    // a proxy id in QNX 4.

    pulse_event = pmsg.event;
    pulserec_rcvid = ctp->rcvid;

    MsgReply(ctp->rcvid, 0, NULL, 0);

    return (_RESMGR_NOREPLY);
}
```

Lastly, when the deliverer process wants to wake up the receiver, it delivers the event. This is analogous to `Trigger(proxy)` in QNX 4.

```
// here is where we send the pulse message.  Note that pulserec_rcvid
// and pulse_event were saved away above.

MsgDeliverEvent (pulserec_rcvid, &pulse_event);
```

Signal services

Here are the fundamental changes to signals:

1 There are a whole bunch of new user-defined signals. These range in value from SIGRTMIN to SIGRTMAX (defined in **`<signal.h>`**). According to POSIX, these can carry data and can be queued.

2 The traditional UNIX signals (SIGINT, SIGHUP, etc.) still exist and in fact are a part of POSIX. According to POSIX, these cannot carry data and cannot be queued. Note, however, that QNX Neutrino doesn't enforce this restriction, so they *can* carry data and be queued.

3 As mentioned above, signals can now be queued. Just as in QNX 4, if a signal is set on a process and that process has the signal blocked (or masked), then the signal is made pending. Unlike QNX 4, however, if the same signal is set on the process a second time while the signal is still blocked, QNX Neutrino can remember that the same signal is now pending *twice*. When the signal is unblocked (or unmasked), then the signal action will take place twice. This is a queued signal. It is set on a signal-by-signal basis and is done at the process level.

The default is that a signal is not queued (i.e. just as in QNX 4). If the signal is set on the process multiple times while the signal is blocked, when it's unblocked the signal will take effect only once. To indicate that a signal is to be queued, set the SA_SIGINFO flag in the *sa_flags* member of the **`struct sigaction`** structure when passing it to *sigaction()*.

4 Signals can also carry data. As with *kill()*, *sigqueue()* can be used to set a signal on a process. Unlike *kill()*, *sigqueue()* has a value parameter. This is data that will be passed to your signal handler. To access that data, your handler function will now have a parameter of type **`siginfo_t`**, which has a member called *si_value*. This will contain the value passed to *sigqueue()*.

Because of the different handler parameters, you must register your handler using *sigaction()*. In QNX 4, you'd put the address of your handler in the *sa_handler* member of the **`struct sigaction`** structure. There is now a new member called *sa_sigaction*. This is where you would put the address of a handler that wanted to get data.

5 You can no longer set SIGKILL and SIGSTOP to be ignored, handled, or blocked (masked).

6 There are issues with multithreaded processes (see the next section).

The *signal(), sigaction(), kill(), sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), siglongjmp(), sigpending(), sigprocmask(), sigsetjmp(),* and *sigsuspend()* functions are used just as in QNX 4. The *sa_flags* member of the **`struct sigaction`** structure that is passed to *sigaction()* now has some flags: SA_NOCLDSTOP and SA_SIGINFO. SA_NOCLDSTOP tells the system not to call the handler if the child is stopped via SIGSTOP (only relevant for the SIGCHLD signal). SA_SIGINFO states that the signal is a queued signal.

There are also some new functions:

- *sigblock()*
- *sigqueue()*
- *sigsetmask()*
- *sigtimedwait()*
- *sigunblock()*
- *sigwait()*
- *sigwaitinfo()*

Signals and threads

Having threads affects how signals are handled. If, for instance, a given process contains six threads, and a signal arrives for the process, which thread is the recipient?

Here are the rules for delivering a signal to a process that has many threads:

- Signal actions are maintained at the process level. If a certain thread decides to ignore or catch a signal, this is remembered at a process level (i.e. not for any specific thread).
- Signal masks, however, are maintained thread-by-thread. If a given thread decides to mask all signals, only that thread is affected.
- The first time a particular signal is set on a process, the kernel looks for a thread that has that signal unmasked. If all threads have the signal masked, then the signal is made pending at the process level until a thread unmasks it, in which case that thread will be affected. If more than one thread has the signal unmasked, then the kernel picks a thread effectively at random. If only one thread has the signal unmasked, then that thread is affected.
- Once a thread as been picked for a particular signal, from then on that particular signal will always go to that thread.

Because of these rules, an easy approach is to dedicate one thread as the “signal-handling thread” and mask signals in all threads except that one. This signal-handling thread could then call *sigwaitinfo()* so as to not consume CPU time while waiting for the signal.

Shared memory

The QNX Neutrino interface to shared memory uses *shm_open()*, *ftruncate()*, *mmap()*, and so on. It is almost the same as newer QNX 4 applications. One major difference is that *ftruncate()* is used, where in QNX 4 you would have used *ltrunc()*. Another difference is that the name of a shared memory object must begin with a slash (`/`) character and contain only one slash to conform to POSIX and to appear in **`/dev/shmem`**. If a name doesn't begin with a slash, it will appear in the current directory. The *shm_ctl()* function is also available for setting additional attributes.

If you're used to calling the *qnx_segment*()* functions, then they'll need to be converted to *shm_open()*, *ftruncate()*, *mmap()* for the new OS. The *qnx_segment*()* functions are no longer supported.

Semaphores

The function calls for semaphores — mainly *sem_init()*, *sem_wait()*, *sem_trywait()*, *sem_post()*, and *sem_destroy()* — are supported in QNX Neutrino with no changes necessary. However, there's the addition of *named semaphores*.

In QNX 4, a semaphore is typically placed into a shared memory area so that two processes can share it. The processes must agree in advance where the semaphore is in memory ("there's a semaphore at offset such-and-such in the shared memory named XYZ"). With QNX Neutrino, and multiple threads, it might make sense for an application to have a semaphore declared locally to a process.

Named semaphores

To make sharing semaphores between processes easier, QNX Neutrino supports POSIX named semaphores. These are semaphores that can be accessed by a name instead of having to be placed in shared memory. For named semaphores to work, you must run the **`mqueue`** process. Named semaphores are created and cleaned up using *sem_open()*, *sem_close()*, and *sem_unlink()*.

Note that *sem_wait()* and *sem_post()* with an unnamed semaphore use kernel calls to do their work, whereas the same functions with a named semaphore work by sending messages to the mqueue process and will be considerably slower.

POSIX Message Queues

The QNX Neutrino interface for POSIX message queues is almost the same as QNX 4. The following are the differences that you'll need to be aware of:

- The name of the message queue server is **`mqueue`** instead of **`Mqueue`**.
- The MQ_ flags are no longer supported.
- The event that you pass to *mq_notify()* has changed in format. Obviously, you can no longer give it a proxy. You can use a pulse instead.

Timers

The timing functions have changed very little:

- *qnx_adj_time()* is now *ClockAdjust()*.
- *clock_setres()* and *qnx_ticksize()* can be replaced with *ClockPeriod()*.
- Many people used the Pentium **`rdtsc`** opcode for getting a cycle counter. A kernel call is now provided that does the same thing — *ClockCycles()*.
- *timer_create()* no longer returns the timer ID. Instead, there's a third parameter that on successful return, contains the timer ID.

Timeouts

Timeouts can now be done using one of two new functions: *TimerTimeout()* or *timer_timeout()*. The only difference between the two is the types of the parameters.

Under QNX 4, a timeout could be achieved by having your blocking function be unblocked by a signal after a certain amount of time elapsed. A problem arises if you're preempted for longer than the timeout. In that case, when your process gets to run again, your signal handler would be called and then you'd enter the blocking function (with no timeout in place, if you didn't use a repeating timer).

Here's a code snippet for a timeout on a *MsgSend()*. Note that we're passing *timer_timeout()* the possible states for *MsgSend()* that we want to timeout.

```
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 );
MsgSend (coid, NULL, 0, NULL, 0);
```

Unfortunately, this still isn't perfect, because the timing is relative to the call to *timer_timeout()*. However, if you're preempted between the *timer_timeout()* and *MsgSend()* for longer than the timeout period, the timeout is still waiting to take place, even though it will be immediate in that case. You also don't have to cancel the timeout, since that will be done automatically before the *MsgSend()* returns.

Interrupts

The only significant change to writing QNX Neutrino interrupt handlers is that the task has been simplified slightly.

You'll find a chapter in the *Programmer's Guide* entitled "Writing an Interrupt Handler." There's also a section in the "Writing a Resource Manager" chapter of that book entitled "Handling interrupts" that shows how to do interrupt handlers from within a resource manager.

Instead of calling *qnx_hint_attach()*, you would call *InterruptAttach()*. The last parameter for *qnx_hint_attach()* was the data segment selector of your process. You no longer need to provide this. The interrupt handler will simply be using the data segment of the process that the handler is in.

In QNX 4, your handler was limited to waking up the process via a proxy. In QNX Neutrino, your handler can return with an event that would contain either a pulse, a signal, or an event of notify type SIGEV_INTR. In the latter case, the attaching thread would block on *InterruptWait()*.

There's also a new way of handling interrupts — with *InterruptAttachEvent()*. In this case you would fill an event with a pulse, signal, or an event of notify type SIGEV_INTR. When the interrupt is generated, the kernel will mask the interrupt and deliver the event, thereby waking up a thread. The thread would then do the required work and then unmask the interrupt. This masking by the kernel is necessary for handling level-sensitive interrupts.

As in QNX 4, you're limited as to which functions you can call from within an interrupt handler. When you look at a function in the library reference manual, one of the areas under the "Classification" heading shows whether or not you can safely call the function from an interrupt handler.

Note that just as in QNX 4 you needed I/O privileges to register an interrupt handler, you still need it under QNX Neutrino. Under the new OS you actually need it for any *Interrupt*()* function except *InterruptWait()*.

To get I/O privileges under QNX 4, you would link with **`-T1`**. To get I/O privileges under QNX Neutrino, you call **`ThreadCtl(_NTO_TCTL_IO, NULL)`**. Note that you must be **`root`** in order to make this call to *ThreadCtl()*.

There are now functions that can be called from both a thread and the interrupt handler for masking and unmasking interrupts: *InterruptMask()* and *InterruptUnmask()*.

Hardware I/O

Port I/O

Port I/O on x86 is done using special machine instructions. On some other platforms, such as PowerPC and MIPS, it's done by mapping in and accessing memory. As such, there's one extra function you need to call that basically works out to a **`NOP`** for x86, but something else on PowerPC and MIPS. That is *mmap_device_io()*. You pass it the number of consecutive ports you want to access and the address of the first port. It simply returns the address of the first port (the same one you gave it). From then on you use instructions such as *in8()*, *out8()*, *in16()*. For addresses, pass them the value returned by *mmap_device_io()* (the base port) plus some offset from this base port.

Note that just as in QNX 4 you needed I/O privileges to do port I/O, you still need I/O privileges under QNX Neutrino.

To get I/O privileges under QNX 4, you would link with **`-T1`**. To get I/O privileges under QNX Neutrino, you call **`ThreadCtl(_NTO_TCTL_IO, NULL)`**. Note that you must be **`root`** in order to make this call to *ThreadCtl()*.

The following is a short example of doing port I/O in QNX Neutrino:

```
#define SERIAL_BASE_PORT 0x2f8
...
#define R_IE    1   /* interrupt enable */
...
#define R_LS    5   /* line status */
...
#define NPORTS  8   /* no. of ports from base port */

uintptr_t iobase;   /* base of io memory (io ports) */

/* initialization, need to do only once */
ThreadCtl (_NTO_TCTL_IO, NULL);
iobase = mmap_device_io (NPORTS, SERIAL_BASE_PORT);

...

/* wait for the transmit holding register to be empty */
while ((in8(iobase + R_LS) & 0x20) == 0)
    ;

/* Enable just the modem status as an interrupt source */
out8 (iobase + R_IE, 0x08);
```

Memory-mapped I/O

When programming for QNX 4, you occasionally need to access physical memory. Typically, this is done for memory-mapped devices (e.g. the PC video RAM). Under QNX Neutrino, the situation is slightly different from QNX 4, but no more complex. There are, moreover, several ways to map physical memory.

The simplest method is to call the QNX Neutrino *mmap_device_memory()* function:

```
virtual_address = mmap_device_memory( NULL, length,
                           PROT_READ | PROT_WRITE | PROT_NOCACHE,
                           MAP_SHARED | MAP_PHYS, physical_address );
```

The above call to *mmap_device_memory()* just does the following:

```
virtual_address = mmap( 0, length,
                        PROT_READ | PROT_WRITE | PROT_NOCACHE,
                        MAP_PHYS | MAP_SHARED, NOFD, physical_address );
```

Note that in neither case do you have to call *shm_open()* as you do in QNX 4.

Memory for DMA

DMA requires that the OS allocate some memory for use by your driver and the DMA controller. You need the virtual address of this memory and the controller needs the physical address. This can all be done using the following code:

```
virtual_address = mmap( 0, length,
                        PROT_READ | PROT_WRITE | PROT_NOCACHE,
                        MAP_PHYS | MAP_ANON, NOFD, 0 );
mem_offset( virtual_address, NOFD, length, &physical_address, 0);
```

Your driver code would use the *virtual_address* and would give *physical_address* to the controller.

PCI functions

QNX 4 has a set of functions whose names begin with *_CA_PCI_*()*. The analogous functions for QNX Neutrino are called *pci_*()*. Note that for QNX Neutrino you must also run a PCI server process (e.g. **pci-bios**). There are no special compile options or stack issues. You also need to call *pci_attach()* to connect to the PCI server before making any other *pci_*()* calls.

Getting system information

The *qnx_osinfo()* function is no longer available. Instead, information can be gathered from a number of places. Not all of the corresponding information that was available from *qnx_osinfo()* is either available or relevant. See the source for the *qnx_osinfo()* function in the migration library to see how to get information for fields for which there is information available. Note that to get a QNX Neutrino-style nodename (the *nodename* member of the **struct _osinfo** structure), you can call *netmgr_ndtostr()*.

Getting process information

Under QNX 4, this was done by repeated calls to *qnx_psinfo()*. As a resource manager in QNX Neutrino, the process manager makes visible the proc filesystem (or **procfs**). If you have a look at the contents of **/proc** you'll see some numbers. These numbers are the process IDs of the processes that are currently executing. To get information on them, you open them and then make *devctl()* calls to the resulting file descriptor. See the source for the *qnx_psinfo()* function in the migration library to see how to do this for a specific process or to walk through all processes. Keep in mind that where under QNX 4 some information would be process-related (e.g. state, blocked on) this information is now thread-related.

The *term_()* Functions

The QNX 4 functions such as *term_delete_char()*, which originated with QNX 2.1, are not supported under QNX Neutrino.

A program using these would need to be reimplemented to use something like curses. Also, the Watcom text calls such as *_gettextcursor()*, and the graphics calls like *_pg_initchart()*, are not supported. Basically, anything in the Watcom Graphics Library Reference is not available in QNX Neutrino.

Chapter 4

Migration Methodology

In this chapter. . .

A suggested approach

This section is intended to discuss one way that you might approach the migration of your applications from QNX 4 to QNX Neutrino. It certainly isn't the only way, and probably not the best way for every migration situation. However, it does offer a starting point for your planning and covers a number of common situations.

Get to know QNX Neutrino

Since you're moving to a new environment, your first step should be to gain some experience with the capabilities of the QNX Neutrino OS. Although this guide points out most of the items you'll have to look after during the migration, there's no substitute for getting your feet wet and finding out exactly how things really work.

The next few sections indicate one way you can build up this experience. If you don't take time to become familiar with the new OS before starting a serious migration project, you'll probably expend much more effort than necessary and have a number of false starts.

Install the OS

The first step is obviously to install your development system. As specified in the "Compiler & tools" section of the Development Environment chapter, your choice of hosts are:

- QNX Neutrino
- MS-Windows
- Solaris

Your migration task will be a lot easier if you can access your target system over the network using NFS or CIFS. This allows you to:

- produce a new executable on your host
- switch over to your target
- run the executable.

Move your environment

The first thing you should do is move your environment. Most of you will have customized the standard QNX environment in many ways with shell scripts and utility programs. Moving this material to QNX Neutrino will give you:

- an initial familiarity with the QNX Neutrino utilities. Many of these will be the same, but some may differ.
- a stable reference point to work from during the migration, one that you are familiar with and feel comfortable with.

Move your utilities

This is where you can start to get a feel for what problems you'll face during the migration of your major applications. Moving your utilities will not only give you a good feel for migration problems, but will also ensure that your standard tools are available to you when the serious work starts.

The section "Moving a program" provides some initial suggestions on how to deal with individual programs.

☞ Keep records on the effort required to migrate each utility. This will give you an estimating base when you come to plan the migration of your major applications.

Plan!

Good planning is the key to a successful migration. The following points provide a starting point for your planning efforts.

- Make an inventory of the third-party software needed for your applications and verify when that software (or equivalent) will be available under QNX Neutrino.
- Establish which applications will migrate to the new OS. It may be that some applications are due for major rewrite and/or upgrade and it may make sense to start with a new QNX Neutrino-based design in some of these cases.
- Establish the order in which applications will migrate. If there are inter-application dependencies, identify them and decide how to deal with them, bearing in mind that applications can communicate between QNX 4 and QNX Neutrino either by moving data from one environment to the other or by IPC over serial links or TCP/IP (but not through native QNX message passing).
- You should develop good estimates of the time required to migrate each application. If you have already converted your utility programs, you'll have a yardstick to help in your estimation of the migration effort.

Moving a program

The following steps will be needed for each program.

Analysis

You can use the **`mig4nto`** utility supplied with the migration kit to identify the major areas in each program that need attention. This utility identifies functions in your source programs that may require attention and provides succinct suggestions as to a course of action in each case. This utility is fully described in the next chapter.

Architectural issues

First you must determine whether, through use of obsolete QNX functions or for other reasons, there are any architectural reasons why the program won't migrate as is. Perhaps the program performs a function that's unique to QNX 4 and isn't necessary in QNX Neutrino. Or maybe a serious redesign is required because the program is too tightly bound to QNX 4 facilities. Programs in this class should be rare, but they do have to be identified and strategies for dealing with them must be developed.

Converting header files

This is largely a mechanical job, but it needs to be done for each program. Many header filenames and contents have changed.

Converting pathnames

If you have hard-coded pathnames in you programs, they may need to be converted. This will be the case with nodename syntax. In QNX 4 a node was denoted using `//`*nid* where *nid* was the network identifier. Nodes of a QNX Neutrino network are different.

Converting functions, etc.

The last step is to convert your program to use QNX Neutrino functions, manifests, and data structures. The output from the **`mig4nto`** utility will be very helpful at this stage.

Chapter 5

The `mig4nto` Utility

In this chapter. . .

mig4nto

Identify items in QNX 4 programs requiring attention (QNX)

Syntax:

mig4nto [-qsr] [-o *output_directory***] [***file***...]**

Options:

-q
: Be quiet; don't display progress information.

-s
: Be strict; allow for only those functions that are ANSI/POSIX-compatible. This option reports on all functions covered by the migration library. If you don't specify this option, functions covered by the migration library aren't identified.

-r
: Produce report only; don't produce a copy of marked-up source.

-o *output_directory*
: The directory where you want annotated source files to be written.

file
: The pathname of a file containing a C source program or header file.

Description

The **mig4nto** utility helps you identify areas in your programs that will need attention during the migration process. This utility copies source files, then inserts comments above each source line that contains a function name or C preprocessor manifest name requiring attention. These comments include brief suggestions as to a course of action.

The marked-up files are written to a directory other than the one from which their corresponding source files were read. You specify this directory with the **-o** option.

For each file it examines, the **mig4nto** utility also provides a summary report listing the items in the file that require attention and the lines on which these items are found. If you want to generate this summary report without also generating annotated program files, then specify the **-r** option.

While processing files, **mig4nto** normally keeps you informed of its progress. But if you want to run the utility in the background, you can specify the **-q** option to prevent status messages from being displayed.

For each input file, **mig4nto** creates an output file in the directory specified by the **-o** option. Each source line containing items that require attention is preceded by a special comment line or lines.

Examples:

Annotate all the C source files in the current directory, and place the output files in the **/nto/src** directory.

```
mig4nto -o /nto/src *.c
```

Annotate all the C source files in the current directory, and place the output in the **/nto/src** directory. Place the summary report in the report file.

```
mig4nto -o /nto/src -s *.c > report
```

Exit status:

0	No files needed attention.
1	One or more files contained items needing attention.
>1	An error occurred.

☞ If a file can't be opened for reading or writing, or if an I/O error occurs while reading or writing a file, the output file is removed and processing is continued with the next file. The final exit status will be greater than one.

Chapter 6

The Migration Library

In this chapter. . .

The migration library is a set of functions that implement many of the QNX 4 functions that are no longer supported or are different in the new OS.

There's also a migration process manager (**`mig4nto-procmgr`**) that must be run for some of the library functions to work.

☞ In the list of migration functions given below, functions that require **`mig4nto-procmgr`** are indicated as such. If a process does call functions that require **`mig4nto-procmgr`**, then you must call the *mig4nto_init()* function at the very beginning of your program. See the description for **`mig4nto-procmgr`** below for more on this.

The migration process manager (`mig4nto-procmgr`)

Provides numerous features of the migration library (QNX)

Syntax:

```
mig4nto-procmgr [-ntv]
```

Options:

-n	QNX 4-style network ID (*nid*) (default: 1)
-t	Number of threads for relaying proxy messages (default: 4)
-v	Be verbose. Use more v's for more information.

Description

The migration process manager provides services such as name registration and name location, proxy registration and sending, and *nid* storage. Basically, it provides things that **`Proc32`** does under QNX 4 but that **`procnto`** (the QNX Neutrino equivalent) doesn't provide (or doesn't provide in the same way).

Triggering of proxies using this library can be much slower than in QNX 4. If a process triggers a proxy and that proxy isn't attached to the triggering process, then the triggerer sends a message to **`mig4nto-procmgr`**. The **`mig4nto-procmgr`** process has a number of threads dedicated to receiving these trigger messages (see the **`-t`** option above). When one of these threads receives the trigger message, it replies back immediately. It then sends a message to the process that the proxy is attached to. The reason for having the *Trigger()* function send to **`mig4nto-procmgr`** is so that it will know if the proxy is a valid one.

Many of the functions in the migration library require that the **`mig4nto-procmgr`** process be running. If using any of these functions, you must also call *mig4nto_init()* at the very beginning of you program, preferably the first thing in *main()*. One of the reasons for this is that *mig4nto_init()* creates a channel by calling *ChannelCreate()*. The channel ID returned *must* be **`1`** — calling this function very early ensures this.

Migration functions that require `mig4nto-procmgr`

The following functions need **`mig4nto-procmgr`** to be running:

- *dev_arm()*
- *getnid()*
- *mig4nto_init()*
- *qnx_name_attach()*
- *qnx_name_detach()*
- *qnx_name_locate()*
- *qnx_name_query()*
- *qnx_proxy_attach()*
- *qnx_proxy_detach()*
- *Readmsg()*
- *Readmsgmx()*
- *Receive()*
- *Receivemx()*
- *Reply()*
- *Replymx()*
- *Send()*
- *Sendmx()*
- *Trigger()*
- *Writemsg()*
- *Writemsgmx()*

The migration library functions

The following functions are in the migration library (**`libmig4nto.a`**). The only include needed is **`<mig4nto.h>`**. Note that full source is available.

Rather than repeat the contents of the QNX 4 documentation, only the differences from QNX 4 implementation are given below. Note that no attempt was made to keep the *errno* failure values the same.

block_read()

The block number has been changed from 1-based to 0-based.

block_write()

The block number has been changed from 1-based to 0-based. The QNX 4 *block_write()*, when applied to a regular file, would never grow it; the QNX Neutrino *writeblock()* function (which this migration function uses) may cause a regular file to be extended if writing occurs beyond the end-of-file.

dev_arm()

Only the following are supported:

- _DEV_EVENT_INPUT
- _DEV_EVENT_OUTPUT
- _DEV_EVENT_EXRDY
- _DEV_EVENT_DRAIN

Requires **`mig4nto-procmgr`** to be running.

dev_info()

If the call is successful, the following members of the info structure are filled:

`int unit`
Unit number of this device (e.g. **`/dev/con2`** would have a unit of 2).

`nid_t nid`
The network ID where this device exists. Note that this will contain a QNX Neutrino node descriptor (*nd*), not a QNX 4 network ID (*nid*).

`pid_t driver_pid`
Process ID of the driver that controls this device.

`char driver_type[16]`
A symbolic name describing the device nature.

`char tty_name[MAX_TTY_NAME]`
A complete pathname that may be used to open his device.

dev_insert_chars()

Same as QNX 4 implementation.

dev_ischars()

Same as QNX 4 implementation.

dev_mode()

_DEV_OSFLOW is not supported.

dev_read()

The proxy and armed parameters are not supported.

dev_readex()

This gets the out-of-band data from **`devc-*`** drivers (uses the DCMD_CHR_GETOBAND of *devctl()*). The return value is the number of bytes read, but is *not* obtained from the driver. Instead, this function estimates the number of bytes by filling the buffer with zeros before doing the *devctl()* and then after the *devctl()* has returned, counting the number of leading non-zero bytes.

dev_size()

Same as QNX 4 implementation.

dev_state()

This only lets you query the current state. The *__bits* and *__mask* parameters are ignored.

The following can be returned:

_DEV_EVENT_INPUT
: Input is available from the device.

_DEV_EVENT_DRAIN
: The output has drained on this device.

_DEV_EVENT_EXRDY
: An exception or out-of-bound character is available to be read with *dev_readex()*

_DEV_EVENT_OUTPUT
: There's room in the output buffer to transmit N chars (by default, N is 1).

disk_get_entry()

This uses *devctl()* with the DCMD_CAM_DEVINFO command. See **`<sys/dcmd_cam.h>`** and **`<sys/cam_device.h>`**.

You may want to use the direct *devctl()*s that build this, because they're more useful and have better field definitions (e.g. "cylinders" in QNX Neutrino is 32-bit, but only 16 in QNX 4; large EIDE disks have already wrapped this due to geometry translation).

disk_space()

It may be better to switch directly to *statvfs()*, which has additional fields that may be useful (such as the block size, the mount flags, etc).

fsys_get_mount_dev()

Same as QNX 4 implementation.

fsys_get_mount_pt()

Same as QNX 4 implementation.

getnid()

This returns the network id passed to **`mig4nto-procmgr`** through the **`-n`** option. The **`mig4nto-procmgr`** process defaults this to 1.

Requires **`mig4nto-procmgr`** to be running.

mig4nto_init()

This must be called before most library functions are called (see the list under the **`mig4nto-procmgr`** section above). It should be called as the first or one of the first things in *main()*. The reason is that it creates a channel for receiving messages on, and the channel ID must be 1. This will be true only for the first call to *ChannelCreate()*. Calling this early ensures that this will be the case.

Requires **`mig4nto-procmgr`** to be running.

qnx_hint_attach()

QNX Neutrino interrupt handlers can return with an event that contains a pulse. QNX 4 interrupt handlers can return a proxy. In order for this to still work, this function installs its own QNX Neutrino interrupt handler that will call the given interrupt handler. So when the interrupt is generated, this *hidden handler* is called. The hidden handler calls the given handler. If the given handler returns with a proxy (non-zero value), then the hidden handler returns a pulse event. The proxy value is stuffed into the event.

The *Receive*()* migration functions watch for this pulse event. When they receive it, they pull the proxy value from the pulse message and return with it.

qnx_hint_detach()

Same as QNX 4 implementation.

qnx_name_attach()

The only valid values for nid are 0 and the local nid (gotten from **`mig4nto-procmgr`**).

Requires **`mig4nto-procmgr`** to be running.

qnx_name_detach()

The only valid values for nid are 0 and the local nid (gotten from **`mig4nto-procmgr`**).

Requires **`mig4nto-procmgr`** to be running.

qnx_name_locate()

The only valid values for nid are 0 and the local nid (gotten from **`mig4nto-procmgr`**).

Requires **`mig4nto-procmgr`** to be running.

qnx_name_query()

The only valid values for proc_pid are 0 and PROC_PID.

Requires **`mig4nto-procmgr`** to be running.

qnx_osinfo()

The fields in the **`osdata`** structure are set to the following values:

- *tick_size* — the current ticksize or resolution of the realtime clock in microseconds.
- *version* — Neutrino 2.0, for example, reports a version of 200, where QNX 4.25 reported 425.
- *sflags* — a bitfield containing:
  - _PSF_PROTECTED — running in protected mode.
  - _PSF_NDP_INSTALLED — FPU hardware is installed.
  - _PSF_EMULATOR_INSTALLED — An FPU emulator is installed
  - _PSF_PCI_BIOS — A PCI BIOS is present
  - _PSF_32BIT_KERNEL — 32-bit kernel is being used.
- *nodename* QNX 4 nid retrieved from the **`mig4nto`** Name Resource Manager.
- *cpu* — processor type (486,586,...)
- *machine* — name of this machine on the network.
- *totpmem* — total physical memory.
- *freepmem* — free physical memory.
- *totmemk* — total memory in Kb, up to USHORT_MAX (65535).
- *freememk* — free memory in Kb, up to USHORT_MAX (65535).
- *cpu_features* — contains CPU speed in MHz.

The remaining fields are set to MIG4NTO_UNSUPP.

qnx_proxy_attach()

Requires **`mig4nto-procmgr`** to be running.

qnx_proxy_detach()

Requires **`mig4nto-procmgr`** to be running.

qnx_psinfo()

Some of the information associated with a process in QNX 4 is associated with a thread in QNX Neutrino (e.g. state, blocked_on, ...). The assumption here is that if you're migrating a QNX 4 process to QNX Neutrino, you'll have only one thread, so this information is taken from the thread with ID 1 (the *main()* thread).

The following fields are populated:

- *pid* — the process ID
- *blocked_on* — what process is blocked on (pid).
- *pid_group* — process group
- *ruid* — real user ID
- *rgid* — real group ID
- *euid* — effective user ID
- *egid* — effective group ID
- *umask* — process umask
- *sid* — session ID
- *signal_ignore* — process signal ignore mask
- *signal_mask* — thread signal block mask
- *state* — thread state
- *priority* — process priority
- *max_priority* — max priority
- *sched_algorithm* — scheduling policy (round-robin or FIFO)
- *un.proc.name* — the name of the program image. This name is limited to 100 bytes including the NULL.
- *un.proc.father* — parent process
- *un.proc.son* — child process
- *un.proc.brother* — sibling process

- *un.proc.times* — all times set to zero

All other **`psdata`** structure elements are set to MIG4NTO_UNSUPP.

QNX Neutrino doesn't support time-accounting information, so the members of the **`tms`** structures are always set to 0.

The *qnx_psinfo()* function can currently examine processes only, as there are no **`/proc`** entries for virtual circuits and pulses.

The only valid values for *proc_pid* are 0 and PROC_PID.

The *segdata* parameter is ignored.

qnx_spawn()

Here are some notes regarding the parameters:

__msgbuf	This is ignored.
__sched_algo	QNX 4 and QNX Neutrino use the same names for scheduling algorithms, but their values are different. Be very careful if you're not just recompiling with the macros from the QNX Neutrino header files. Note also that in the new OS, SCHED_OTHER is SCHED_RR. QNX Neutrino doesn't have QNX 4's adaptive scheduling algorithm. As such, there's no equivalent of SCHED_FAIR.
__flags	The _SPAWN_XCACHE flag is not supported.
__iov	If this is given, then unlike QNX 4, the FDs passed within it will be the only ones inherited by the child. This is true even for the IOVs that are -1.
__ctfd	This returns -1 and sets errno to EINVAL if the *__ctfd* parameter is anything other than -1.

Readmsg()

Requires **`mig4nto-procmgr`** to be running.

Readmsgmx()

Requires **`mig4nto-procmgr`** to be running.

Receive()

Requires **`mig4nto-procmgr`** to be running.

Receivemx()

Requires **`mig4nto-procmgr`** to be running.

Reply()

Requires **`mig4nto-procmgr`** to be running.

Replymx()

Requires **`mig4nto-procmgr`** to be running.

Send()

This function creates a connection for each process that is sent to. Once the message has been sent, the connection is *not* detached. Instead, the connection ID is cached in case further messages are sent to the same process.

Requires **`mig4nto-procmgr`** to be running.

Sendmx()

This function creates a connection for each process that is sent to. Once the message has been sent, the connection is *not* detached. Instead, the connection ID is cached in case further messages are sent to the same process.

Requires **`mig4nto-procmgr`** to be running.

Trigger()

If triggering a proxy that is attached to the calling process, then this uses a pulse.

If triggering a proxy that is attached to another process, then this sends a message to **`mig4nto-procmgr`**, which has a set of dedicated threads for receiving this message. When one of those threads receives the message, it replies immediately and then sends a message to the process that the proxy is attached to.

The reason for going through **`mig4nto-procmgr`** is that the *Trigger()* function will at least know whether or not the proxy is a valid one.

Requires **`mig4nto-procmgr`** to be running.

Writemsg()

Requires **`mig4nto-procmgr`** to be running.

Writemsgmx()

Requires **`mig4nto-procmgr`** to be running.

Yield()

Same as QNX 4 implementation.

Appendix A

QNX 4 Functions & QNX Neutrino Equivalents

This appendix lists the QNX 4 C library functions along with their QNX Neutrino equivalents. For functions that have no direct replacement, you'll find either a cover function or a suggested workaround.

abstimer()

QNX Neutrino equivalent:

`timer_settime( CLOCK_REALTIME, TIMER_ABSTIME, ...  )`

In migration library?

No

_asctime()

QNX Neutrino equivalent:

extern char *asctime_r(const struct tm *__*timeptr***, char *__***buff*** **);**

In migration library?

No

This call is a drop-in replacement.

_bcalloc()

QNX Neutrino equivalent:

calloc()

In migration library?

No

QNX Neutrino doesn't support segment-based functions.

_beginthread()

QNX Neutrino equivalent:

pthread_create()

In migration library?

No

☞ A thread in QNX 4 is really just a separate process that shares the data segment of its parent, whereas a thread in QNX Neutrino is really within *the same process* as its parent and shares a great deal more.

_bexpand()

QNX Neutrino equivalent:

realloc()

In migration library?

No

☞ QNX Neutrino doesn't support segment-based functions. You can use *realloc()* in place of this, but beware that *realloc()* will move your memory block to a new address if needed, and *_bexpand()* will fail rather than move your memory block to a new address.

_bfree()

QNX Neutrino equivalent:

free()

In migration library?

No

QNX Neutrino doesn't support segment-based functions.

_bfreeseg()

QNX Neutrino equivalent:

free()

In migration library?

No

QNX Neutrino doesn't support segment-based functions.

_bgetcmd()

QNX Neutrino equivalent:

Parse the argument vector passed to *main()* instead.

In migration library?

No

QNX Neutrino doesn't support segment-based functions.

_bheapchk()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_bheapmin()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_bheapseg()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_bheapset()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_bheapshrink()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_bheapwalk()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the **malloc_g** library described in the “Heap Analysis” chapter in *Programmer’s Guide*.

block_read()

QNX Neutrino equivalent:

readblock() as follows:

```
readblock(filedes, 512, block - 1, nblocks, buf)
```

In migration library?

Yes

☞ The block number has been changed from 1-based to 0-based.

block_write()

QNX Neutrino equivalent:

writeblock() as follows:

```
writeblock(filedes, 512, block - 1, nblocks, buf)
```

In migration library?

Yes

☞ The block number has been changed from 1-based to 0-based. When applied to a regular file, the QNX 4 *block_write()* would never grow the file; the *writeblock()* function may cause a regular file to be extended if writing occurs beyond the end-of-file. Note that the cover function in the migration library calls *writeblock()*.

_bmalloc()

QNX Neutrino equivalent:

malloc()

In migration library?

No

QNX Neutrino doesn't support segment-based functions.

_bmsize()

QNX Neutrino equivalent:

_msize(), *_musize()*, and *DH_ULEN()*

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_bprintf()

QNX Neutrino equivalent:

snprintf()

In migration library?

No

_brealloc()

QNX Neutrino equivalent:

realloc()

In migration library?

No

QNX Neutrino doesn't support segment-based functions.

*_CA_PCI_** functions

The following functions aren't in the migration library:

QNX 4 function:	QNX Neutrino equivalent:
_CA_PCI_BIOS_Present()	*pci_present()*

continued...

QNX 4 function:	QNX Neutrino equivalent:
_CA_PCI_Find_Class()	*pci_find_class()*
_CA_PCI_Find_Device()	*pci_find_device()*
_CA_PCI_Generate_Special_Cycle()	No longer supported
_CA_PCI_Read_Config_Byte()	*pci_read_config8()*
_CA_PCI_Read_Config_DWord()	*pci_read_config32()*
_CA_PCI_Read_Config_Word()	*pci_read_config16()*
_CA_PCI_Write_Config_Byte()	*pci_write_config8()*
_CA_PCI_Write_Config_DWord()	*pci_write_config32()*
_CA_PCI_Write_Config_Word()	*pci_write_config16()*

cgets()

QNX Neutrino equivalent:

Set **`/dev/tty`** as standard output and call *gets()*.

In migration library?

No

_clear87()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

clock_setres()

QNX Neutrino equivalent:

ClockPeriod()

In migration library?

No

console_active()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_arm()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_close()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_ctrl()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_info()

QNX Neutrino equivalent:

tcgetsize() for the number of rows and columns — the remainder is no longer supported.

In migration library?

No

console_open()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_protocol()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_read()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_size()

QNX Neutrino equivalent:

tcgetsize() for the number of rows and columns — you can't set the size.

In migration library?

No

console_state()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

console_write()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_control87()

QNX Neutrino equivalent:

See **`<fpstatus.h>`**

In migration library?

No

cprintf()

QNX Neutrino equivalent:

Set **`/dev/tty`** as standard output and call *printf()*.

In migration library?

No

cputs()

QNX Neutrino equivalent:

Do *fputs()* to **`/dev/tty`** instead.

In migration library?

No

Creceive()

QNX Neutrino equivalent:

MsgReceive() preceded immediately by:

```
event.sigev_notify = SIGEV_UNBLOCK;
TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_RECEIVE,
            &event, NULL, NULL );
```

In migration library?

No

Creceivemx()

QNX Neutrino equivalent:

MsgReceivev() preceded immediately by:

```
event.sigev_notify = SIGEV_UNBLOCK;
TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_RECEIVE,
            &event, NULL, NULL );
```

In migration library?

No

crypt()

QNX Neutrino equivalent:

crypt()

In migration library?

No

The QNX Neutrino version is Unix-compatible. For the QNX 4 version, *qnx_crypt()*.

cscanf()

QNX Neutrino equivalent:

Set **`/dev/tty`** as standard input and call *scanf()*.

In migration library?

No

_ctime()

QNX Neutrino equivalent:

ctime_r()

In migration library?

No

cuserid()

QNX Neutrino equivalent:

geteuid() for the user ID number followed by *getpwent()* to find the user name.

In migration library?

No

dev_arm()

QNX Neutrino equivalent:

See *ionotify()*.

In migration library?

Yes — covers _DEV_EVENT_INPUT, _DEV_EVENT_OUTPUT, _DEV_EVENT_EXRDY, and _DEV_EVENT_DRAIN.

Not all event types are supported. For _DEV_EVENT_HANGUP, consider setting up a controlling terminal and handle SIGHUP. For _DEV_EVENT_WINCH, consider using SIGWINCH.

dev_fdinfo()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

You can't get all the information, but you can get bits and pieces elsewhere.

dev_info()

QNX Neutrino equivalent:

No longer supported.

In migration library?

Yes

You can't get all the information, but you can get bits and pieces elsewhere.

dev_insert_chars()

QNX Neutrino equivalent:

tcinject()

In migration library?

Yes

dev_ischars()

QNX Neutrino equivalent:

tcischars()

In migration library?

Yes

dev_mode()

QNX Neutrino equivalent:

tcgetattr() and *tcsetattr()*

In migration library?

Yes

dev_osize()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

dev_read()

QNX Neutrino equivalent:

readcond() and *ionotify()*

In migration library?

Yes

You can implement some of this by using *readcond()* and *ionotify()*. The *readcond()* can handle the cases where **`proxy == 0`**. The *ionotify()* in conjunction with pulses (signals are even easier) can handle the cases where **`proxy != 0`** and *min*, *time*, and *timeout* are all **`0`**.

For equivalent functionality to *min*, *time*, and *timeout* combined with a pulse or signal for notification, create a separate thread that requests pulse notification using *ionotify()*. Set up another pulse notification for the timeout. Then go into a *MsgReceive()* loop with with a *TimerTimeout()* call before the *MsgReceive()* call for the interbyte time. Deliver a pulse or set a signal when the *min*, *time* or *timeout* condition is satisfied.

☞ The cover function doesn't handle the proxy and armed parameters.

dev_readex()

QNX Neutrino equivalent:

devctl() with DCMD_CHR_GETOBAND

In migration library?

Yes

You can use *devctl()* with DCMD_CHR_GETOBAND in place of this for getting out-of-band data from resource managers that support it. Currently, only **`devc-*`** resource managers support this.

dev_size()

QNX Neutrino equivalent:

tcgetsize() and *tcsetsize()*

In migration library?

Yes

dev_state()

QNX Neutrino equivalent:

N/A

In migration library?

Yes

There's no equivalent way of directly setting these states:

- For _DEV_EVENT_INPUT, use *devctl()* with DCMD_CHR_ISCHARS.
- For _DEV_EVENT_DRAIN, use *devctl()* with DCMD_CHR_OSCHARS.

- For _DEV_EVENT_OUTPUT, there's currently no way to determine this (because there's no way to determine the size of the output buffer).
- For _DEV_EVENT_EXRDY, use *devctl()* with DCMD_CHR_GETOBAND — note that doing so will clear the out-of-band data in the case of **devc-*** drivers.

_disable()

QNX Neutrino equivalent:

InterruptLock()

In migration library?

No

disk_get_entry()

QNX Neutrino equivalent:

devctl() with the **DCMD_CAM_DEVINFO** command.

In migration library?

Yes, but see below.

See **<sys/dcmd_cam.h>** and **<sys/cam_device.h>**. Although a cover function is provided in the migration library, you might want to use the direct *devctl()*s that build this — they're more useful and have better field definitions (e.g. "cylinders" in QNX Neutrino is 32-bit, but only 16-bit in QNX 4, and large EIDE disks have already wrapped this due to geometry translation).

disk_space()

QNX Neutrino equivalent:

statvfs()

In migration library?

Yes, but see below.

Although a cover function is provided in the migration library, you might want to switch directly to *statvfs()* because it has additional fields that may be useful (block size, mount flags, etc.).

ecvt()

QNX Neutrino equivalent:

sprintf()

In migration library?

No

_ecvt()

QNX Neutrino equivalent:

sprintf()

In migration library?

No

_enable()

QNX Neutrino equivalent:

InterruptUnlock()

In migration library?

No

_endthread()

QNX Neutrino equivalent:

pthread_exit()

In migration library?

No

_expand()

QNX Neutrino equivalent:

realloc()

In migration library?

No

☞ You can use *realloc()* in place of this, but beware that *realloc()* will move your memory block to a new address if needed, and *_expand()* will fail rather than move your memory block to a new address.

_fcalloc()

QNX Neutrino equivalent:

calloc()

In migration library?

No

fcvt()

QNX Neutrino equivalent:

sprintf()

In migration library?

No

_fcvt()

QNX Neutrino equivalent:

sprintf()

In migration library?

No

_fexpand()

QNX Neutrino equivalent:

realloc()

In migration library?

No

You can use *realloc()* in place of this, but beware that *realloc()* will move your memory block to a new address if needed, and *_expand()* will fail rather than move your memory block to a new address.

_ffree()

QNX Neutrino equivalent:

free()

In migration library?

No

_fheapchk()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_fheapgrow()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_fheapmin()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_fheapset()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **`malloc_g`** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_fheapshrink()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_fheapwalk()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the **`malloc_g`** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

filelength()

QNX Neutrino equivalent:

fstat()

In migration library?

No

You can do this with:

```
fstat(fd, &st);
```

followed by:

```
return(S_ISBLK(st.st_mode) ? st.st_nblocks *
       st.st_blocksize : st.st_size);
```

A little more may be needed for 64-bit support.

_fmalloc()

QNX Neutrino equivalent:

malloc()

In migration library?

No

_fmemccpy()

QNX Neutrino equivalent:

memccpy()

In migration library?

No

_fmemchr()

QNX Neutrino equivalent:

memchr()

In migration library?

No

_fmemcmp()

QNX Neutrino equivalent:

memcmp()

In migration library?

No

_fmemcpy()

QNX Neutrino equivalent:

memcpy()

In migration library?

No

_fmemicmp()

QNX Neutrino equivalent:

memicmp()

In migration library?

No

_fmemmove()

QNX Neutrino equivalent:

memmove()

In migration library?

No

_fmemset()

QNX Neutrino equivalent:

memset()

In migration library?

No

_fmsize()

QNX Neutrino equivalent:

_msize(), *_musize()*, and *DH_ULEN()*

In migration library?

No

See the **`malloc_g`** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

FP_OFF()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_fpreset()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_frealloc()

QNX Neutrino equivalent:

realloc()

In migration library?

No

_freect()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

FP_SEG()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_fsopen()

QNX Neutrino equivalent:

sopen(), *fdopen()*

In migration library?

No

Use *sopen()*, which returns a file descriptor, then use *fdopen()* to associate a stream with it.

*_fstr** functions

The following functions aren't in the migration library:

QNX 4 function:	QNX Neutrino equivalent:
_fstrcat()	*strcat()*
_fstrchr()	*strchr()*
_fstrcmp()	*strcmp()*
_fstrcpy()	*strcpy()*
_fstrcspn()	*strcspn()*
_fstrdup()	*strdup()*
_fstricmp()	*strcmp()*
_fstrlen()	*strlen()*
_fstrlwr()	*strlwr()*
_fstrncat()	*strncat()*
_fstrncmp()	*strncmp()*
_fstrncpy()	*strncpy()*
_fstrnicmp()	*strnicmp()*
_fstrnset()	*strnset()*
_fstrpbrk()	*strpbrk()*
_fstrrchr()	*strrchr()*
_fstrrev()	*strrev()*
_fstrset()	*strset()*
_fstrspn()	*strspn()*
_fstrstr()	*strstr()*
_fstrtok()	*strtok()*
_fstrupr()	*strupr()*

fsys_fdinfo()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

fsys_fstat()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

The filesystem doesn't provide the information because the QNX Neutrino **io-blk** doesn't give preferential treatment to any particular disk format.

fsys_get_mount_dev()

QNX Neutrino equivalent:

devctl()

In migration library?

Yes

You can use the *devctl()* command DCMD_FSYS_MOUNTED_ON to get this information, but it must be sent as part of a combine message. See the source for *fsys_get_mount_dev()* in the migration library for code for doing this.

fsys_get_mount_pt()

QNX Neutrino equivalent:

devctl()

In migration library?

Yes

You can use the *devctl()* command DCMD_FSYS_MOUNTED_BY to get this information, but it must be sent as part of a combine message. See the source for *fsys_get_mount_pt()* in the migration library for code for doing this.

fsys_stat()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

The filesystem doesn't provide the information because the QNX Neutrino **io-blk** doesn't give preferential treatment to any particular disk format.

gcvt()

QNX Neutrino equivalent:

Consider *sprintf()*

In migration library?

No

_gcvt()

QNX Neutrino equivalent:

  Consider *sprintf()*

In migration library?

  No

getch()

QNX Neutrino equivalent:

  read() in raw mode.

In migration library?

  No

getche()

QNX Neutrino equivalent:

  getchar() or *getc()* combined with *putchar()* or *putc()*.

In migration library?

  No

getcmd()

QNX Neutrino equivalent:

  Parse the argument vector passed to *main()* instead.

In migration library?

  No

getnid()

QNX Neutrino equivalent:

  netmgr_ndtostr()

In migration library?

  Yes

Unlike QNX 4, QNX Neutrino doesn't use node IDs (*nid*s). Instead, nodes on a network have *names*. To get the name of the caller's node, use *netmgr_ndtostr()* with the nd parameter set to ND_LOCAL_NODE.

The migration library has a *getnid()* function that returns whatever was passed to **`mig4nto-procmgr`** via the **`-n`** option.

gettimer()

QNX Neutrino equivalent:

timer_gettime()

In migration library?

No

getwd()

QNX Neutrino equivalent:

getcwd()

In migration library?

No

The *getwd()* function requires a preallocated buffer, whereas *getcwd()* will allocate one if it's passed NULL for the buffer. The *getcwd()* function also has a size parameter. For portability, use *getcwd()* instead of *getwd()*.

_gmtime()

QNX Neutrino equivalent:

gmtime_r()

In migration library?

No

halloc()

QNX Neutrino equivalent:

calloc()

In migration library?

No

_heapchk()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **`malloc_g`** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_heapenable()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_heapgrow()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_heapmin()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_heapset()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **`malloc_g`** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_heapshrink()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_heapwalk()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the **`malloc_g`** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

hfree()

QNX Neutrino equivalent:

free()

In migration library?

No

ioctl()

QNX Neutrino equivalent:

devctl()

In migration library?

No

For the **`dcmd`**s to use with QSS-supplied resource managers, see **`<sys/dcmd_*.h>`**.

inp()

QNX Neutrino equivalent:

in8()

In migration library?

No

☞ You should call *mmap_device_io()* before calling the port I/O functions.

inpd()

QNX Neutrino equivalent:

in32()

In migration library?

No

☞ You should call *mmap_device_io()* before calling the port I/O functions.

inpw()

QNX Neutrino equivalent:

in16()

In migration library?

No

☞ You should call *mmap_device_io()* before calling the port I/O functions.

__isascii()

QNX Neutrino equivalent:

isascii()

In migration library?

No

__iscsym()

QNX Neutrino equivalent:

isalpha(), *isdigit()*

In migration library?

No

Replace with an expression using *isalpha()*, *isdigit()* and testing for the underscore character.

__iscsymf()

QNX Neutrino equivalent:

isalpha()

In migration library?

No

Replace with an expression using *isalpha()* and testing for the underscore character.

_itoa()

QNX Neutrino equivalent:

itoa()

In migration library?

No

kbhit()

QNX Neutrino equivalent:

tcischars()

In migration library?

No

lfind()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_localtime()

QNX Neutrino equivalent:

localtime_r()

In migration library?

No

lock()

QNX Neutrino equivalent:

fcntl() with F_SETLK

In migration library?

No

locking()

QNX Neutrino equivalent:

tell() and *fcntl()* with F_SETLK

In migration library?

No

_locking()

QNX Neutrino equivalent:

tell() and *fcntl()* with F_SETLK

In migration library?

No

log2()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_lrotl()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_lrotr()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

lsearch()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_ltoa()

QNX Neutrino equivalent:

ltoa()

In migration library?

No

_makepath()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

__max()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_memavl()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_memicmp()

QNX Neutrino equivalent:

memicmp()

In migration library?

No

_memmax()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

__min()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

MK_FP()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

mktimer()

QNX Neutrino equivalent:

timer_create()

In migration library?

No

mount()

QNX Neutrino equivalent:

mount()

In migration library?

No

☞ This is supported, but its prototype has changed.

mouse_close()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

mouse_flush()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

mouse_open()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

mouse_param()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

mouse_read()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

movedata()

QNX Neutrino equivalent:

memcpy()

In migration library?

No

_msize()

QNX Neutrino equivalent:

_msize(), *_musize()*, and *DH_ULEN()*

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_ncalloc()

QNX Neutrino equivalent:

calloc()

In migration library?

No

_nexpand()

QNX Neutrino equivalent:

realloc()

In migration library?

No

You can use *realloc()* in place of this, but beware that *realloc()* will move your memory block to a new address if needed, and *_nexpand()* will fail rather than move your memory block to a new address.

_nfree()

QNX Neutrino equivalent:

free()

In migration library?

No

_nheapchk()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_nheapgrow()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_nheapmin()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_nheapset()

QNX Neutrino equivalent:

mallopt() with MALLOC_VERIFY

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_nheapshrink()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_nheapwalk()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

_nmalloc()

QNX Neutrino equivalent:

malloc()

In migration library?

No

_nmsize()

QNX Neutrino equivalent:

_msize(), *_musize()*, and *DH_ULEN()*

In migration library?

No

See the **malloc_g** library described in the "Heap Analysis" chapter in *Programmer's Guide*.

nosound()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_nrealloc()

QNX Neutrino equivalent:

realloc()

In migration library?

No

onexit()

QNX Neutrino equivalent:

atexit()

In migration library?

No

outp()

QNX Neutrino equivalent:

out8()

In migration library?

No

☞ You should call *mmap_device_io()* before calling the port I/O functions.

outpd()

QNX Neutrino equivalent:

out32()

In migration library?

No

☞ You should call *mmap_device_io()* before calling the port I/O functions.

outpw()

QNX Neutrino equivalent:

out16()

In migration library?

No

☞ You should call *mmap_device_io()* before calling the port I/O functions.

print_usage()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

putch()

QNX Neutrino equivalent:

Set **`/dev/tty`** as standard output and call *putchar()*.

In migration library?

No

Alternate method: simply open **`/dev/tty`** and use *putc()*.

qnx_adj_time()

QNX Neutrino equivalent:

ClockAdjust()

In migration library?

No

qnx_device_attach()

QNX Neutrino equivalent:

rsrcdbmgr_devno_attach()

In migration library?

No

qnx_device_detach()

QNX Neutrino equivalent:

rsrcdbmgr_devno_detach()

In migration library?

No

qnx_display_hex()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_display_msg()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_fd_attach()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

The resource manager library removes the need for this.

qnx_fd_detach()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

The resource manager library removes the need for this.

qnx_fd_query()

QNX Neutrino equivalent:

None currently

In migration library?

No

There may be something later.

qnx_fullpath()

QNX Neutrino equivalent:

realpath()

In migration library?

No

Use *realpath()* followed by a call to *netmgr_ndtostr()* to get the node name.

qnx_getclock()

QNX Neutrino equivalent:

None currently for remote nodes.

In migration library?

No

Currently, there's no way of getting the time from another node in a native QNX network. Use *clock_gettime()* to get the time on the local node.

qnx_getids()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the section on “Getting process information” in the Programming Issues chapter in this guide.

qnx_hint_attach()

QNX Neutrino equivalent:

InterruptAttach() or *InterruptAttachEvent()*

In migration library?

No

qnx_hint_detach()

QNX Neutrino equivalent:

InterruptDetach()

In migration library?

No

qnx_hint_mask()

QNX Neutrino equivalent:

InterruptMask() and *InterruptUnmask()*

In migration library?

No

qnx_hint_query()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_ioctl()

QNX Neutrino equivalent:

devctl()

In migration library?

No

For the **`dcmd`**s to use with QSS-supplied resource managers, see **`<sys/dcmd_*.h>`**.

qnx_ioctlmx()

QNX Neutrino equivalent:

devctl()

In migration library?

No

For the **`dcmd`**s to use with QSS-supplied resource managers, see **`<sys/dcmd_*.h>`**.

qnx_name_attach()

QNX Neutrino equivalent:

name_attach() or write a resource manager

In migration library?

Yes

If you're not using the migration library and you're using QNX Neutrino, then use *name_attach()* or write resource managers.

For some other methods that the sender can use to find the receiver, see the section on "How does the sender find the receiver?" in the Programming Issues chapter in this guide.

qnx_name_detach()

QNX Neutrino equivalent:

name_detach() or write a resource manager

In migration library?

Yes

If you're not using the migration library and you're replacing *qnx_name_attach()* with *name_attach()*, then use *name_detach()*.

qnx_name_locate()

QNX Neutrino equivalent:

name_open() or write a resource manager

In migration library?

Yes

If you're not using the migration library and you're using QNX Neutrino, then use *name_attach()* or write resource managers.

For some other methods that the sender can use to find the receiver, see the section on "How does the sender find the receiver?" in the Programming Issues chapter in this guide.

qnx_name_locators()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_name_nodes()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_name_query()

QNX Neutrino equivalent:

Names registered via *name_attach()* (QNX Neutrino) appear in **`/dev/name/local`** and **`/dev/name/global`**.

In migration library?

Yes

The migration library has a *qnx_name_query()* function for querying the names registered using the *qnx_name_attach()* migration library function.

qnx_net_alive()

QNX Neutrino equivalent:

netmgr_ndtostr() with ND_LOCAL_NODE, *readdir()*

In migration library?

No

Find out the name of your network directory by calling *netmgr_ndtostr()* with ND_LOCAL_NODE for the *nd* parameter. Then walk through the network directory using *readdir()*. The nodes listed are those that are up.

qnx_nidtostr()

QNX Neutrino equivalent:

netmgr_ndtostr()

In migration library?

No

qnx_osinfo()

QNX Neutrino equivalent:

No longer supported.

In migration library?

Yes

See the section on "Getting system information" in the Programming Issues chapter in this guide.

qnx_osstat()

QNX Neutrino equivalent:

sysconf()

In migration library?

No

QNX Neutrino doesn't have as many hard limits as QNX 4, but instead keeps allocating memory until it runs out. Some limits can be found out by calling *sysconf()*.

qnx_pflags()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the section on "Process flags" in the Programming Issues chapter in this guide.

qnx_prefix_attach()

QNX Neutrino equivalent:

name_attach() or *pathmgr_symlink()*

In migration library?

No

If you were using this function just to put a name in the prefix table so that other processes could find yours, then use *name_attach()* instead (QNX Neutrino).

If you were using this function in an I/O manager that handled **_IO_*** messages, then you need to convert to the resource manager library.

If you were using this function to create an alias, then use *pathmgr_symlink()* instead.

qnx_prefix_detach()

QNX Neutrino equivalent:

name_detach() or *resmgr_detach()* or *pathmgr_unlink()* or *unlink()*

In migration library?

No

If you're using *name_attach()* to register a name (QNX Neutrino), then use *name_detach()* to detach it.

If you're writing a resource manager and had attached the name via *resmgr_attach()*, then use *resmgr_detach()* to detach it.

If you wanted to remove a symlink created using *pathmgr_symlink()*, then use *pathmgr_unlink()* or *unlink()* instead.

qnx_prefix_getroot()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

QNX Neutrino doesn't have the concept of a network root.

qnx_prefix_query()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

For the names that are associated with a resource manager, you can walk through the directory structure under **`/proc/mount`**. The numbers shown refer to a resource manager and are *nd*,*pid*,*chid*,*handle*,*type* where the *type* is one of the _FTYPE_* macros in **`<sys/ftype.h>`**. Names that are the equivalent of replacements (or aliases) aren't visible in QNX Neutrino.

qnx_prefix_setroot()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

QNX Neutrino doesn't have the concept of a network root.

qnx_proxy_attach()

QNX Neutrino equivalent:

Replace proxies with pulses

In migration library?

Yes

If you're not using the migration library, then consider replacing proxies with pulses.

qnx_proxy_detach()

QNX Neutrino equivalent:

No longer supported.

In migration library?

Yes

If you're not using the migration library and you're replacing *qnx_proxy_attach()* with pulses, then you may need to detach the connection for delivering the pulse.

qnx_proxy_rem_attach()

QNX Neutrino equivalent:

Replace proxies with pulses

In migration library?

No

If you're not using the migration library, then consider replacing proxies with pulses.

qnx_proxy_rem_detach()

QNX Neutrino equivalent:

Replace proxies with pulses

In migration library?

No

If you're not using the migration library and you're replacing *qnx_proxy_rem_attach()* with pulses, then you may need to detach the connection for delivering the pulse.

qnx_psinfo()

QNX Neutrino equivalent:

No longer supported.

In migration library?

Yes

See the section on "Getting system information" in the Programming Issues chapter in this guide.

qnx_scheduler()

QNX Neutrino equivalent:

sched_setscheduler() for the local case.

In migration library?

No

Currently, there's no way to do this across the network.

qnx_segment_alloc()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See *shm_open()*, *ftruncate()*, and *mmap()*.

qnx_segment_alloc_flags()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See *shm_open()*, *ftruncate()*, and *mmap()*.

qnx_segment_arm()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_segment_flags()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See *mmap()*.

qnx_segment_free()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See *munmap()* and *shm_unlink()*.

qnx_segment_get()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_segment_huge()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_segment_index()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_segment_info()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

To get a physical address, use *posix_mem_offset()*.

qnx_segment_overlay()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See *mmap_device_memory()* or *mmap()*.

qnx_segment_overlay_flags()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See *mmap_device_memory()* or *mmap()*.

qnx_segment_put()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_segment_raw_alloc()

QNX Neutrino equivalent:

shm_ctl()

In migration library?

No

Create a shared memory object and use *shm_ctl()* to both set its size and to create it as contiguous. If the process dies, then as long as you don't do *shm_unlink()* the memory will still be set aside. To get a physical address, use *posix_mem_offset()*.

qnx_segment_raw_free()

QNX Neutrino equivalent:

shm_unlink()

In migration library?

No

To return memory allocated as detailed above under *qnx_segment_raw_alloc(), close()* the file descriptor, *munmap()* the memory, and call *shm_unlink()*.

There's no equivalent function for adding memory that wasn't reported by the BIOS. However, this sort of thing can be done using the **`-M`** option to **`startup-*`** (*Utilities* reference) or from the startup code using *add_mem()* (see *Building Embedded Systems* in the Embedding SDK package).

qnx_segment_realloc()

QNX Neutrino equivalent:

N/A

In migration library?

No

You can grow shared memory at any time. You can shrink it only to 0 bytes and only when no other process has it mapped. Shrinking it to other sizes may be implemented in a future release.

qnx_setclock()

QNX Neutrino equivalent:

clock_settime() for the local case.

In migration library?

No

Currently, there's no way to do this across the network.

qnx_setids()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_sflags()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

To obtain the equivalent of most of these flags, see the source for *qnx_osinfo()* in the migration library.

qnx_sid_query()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

qnx_spawn()

QNX Neutrino equivalent:

spawn() and *spawn** family.

In migration library?

Yes

See the section on “Process issues” in the Programming Issues chapter in this guide.

qnx_strtonid()

QNX Neutrino equivalent:

netmgr_ndtostr()

In migration library?

No

qnx_sync()

QNX Neutrino equivalent:

sync(), possibly with *fdatasync()* or *fsync()*.

In migration library?

No

☞ These functions don’t synchronize a filesystem on another node of the network.

qnx_ticksize()

QNX Neutrino equivalent:

ClockPeriod()

In migration library?

No

This behaves like *qnx_ticksize()* with the _TICKSIZE_CLOSEST flag.

qnx_trace_close()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Consider using *syslog()* for logging instead.

qnx_trace_info()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Consider using *syslog()* for logging instead.

qnx_trace_open()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Consider using *syslog()* for logging instead.

qnx_trace_read()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Consider using *syslog()* for logging instead.

qnx_trace_severity()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Consider using *syslog()* for logging instead.

qnx_trace_trigger()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Consider using *syslog()* for logging instead.

qnx_umask()

QNX Neutrino equivalent:

N/A

In migration library?

No

There's no way to set the **`umask`** of another process, but you can use *umask()* to set the **`umask`** for the caller.

qnx_vc_attach()

QNX Neutrino equivalent:

ConnectAttach()

In migration library?

No

qnx_vc_detach()

QNX Neutrino equivalent:

ConnectDetach()

In migration library?

No

qnx_vc_name_attach()

QNX Neutrino equivalent:

open() or *name_open()*

In migration library?

No

This is the equivalent of doing *open()* (or *name_open()*) of a name that is registered by a process on another node via *resmgr_attach()* (or *name_attach()*).

qnx_vc_poll_parm()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

See the various docs on QNX Neutrino native networking for similar options.

Readmsg()

QNX Neutrino equivalent:

MsgRead()

In migration library?

Yes

☞ Call this function with the receive ID returned from *MsgReceive()* instead of a process ID.

Readmsgmx()

QNX Neutrino equivalent:

MsgReadv()

In migration library?

Yes

☞ Call this function with the receive ID returned from *MsgReceive()* instead of a process ID.

Receive()

QNX Neutrino equivalent:

MsgReceive()

In migration library?

Yes

☞ Call this function with a channel ID returned from *ChannelCreate()* instead of a process ID.

For more information, see the section on “Channel IDs vs process IDs” in the Programming Issues chapter in this guide.

Receivemx()

QNX Neutrino equivalent:

MsgReceivev()

In migration library?

Yes

☞ Call this function with a channel ID returned from *ChannelCreate()* instead of a process ID.

For more information, see the section on “Channel IDs vs process IDs” in the Programming Issues chapter in this guide.

Relay()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

reltimer()

QNX Neutrino equivalent:

```
timer_settime( CLOCK_REALTIME, 0, ...)
```

In migration library?

No

Reply()

QNX Neutrino equivalent:

MsgReply()

In migration library?

Yes

☞ Call this function with the receive ID returned from *MsgReceive()* instead of a process ID.

Replymx()

QNX Neutrino equivalent:

MsgReplyv()

In migration library?

Yes

☞ Call this function with the receive ID returned from *MsgReceive()* instead of a process ID.

rmtimer()

QNX Neutrino equivalent:

timer_delete()

In migration library?

No

_rotl()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_rotr()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_searchenv()

QNX Neutrino equivalent:

searchenv()

In migration library?

No

☞ The *searchenv()* function doesn't search in the current directory unless it's specified in the given environment variable.

segread()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

Send()

QNX Neutrino equivalent:

MsgSend()

In migration library?

Yes

☞ This function takes a connection ID (*coid*) instead of a process ID. You can get this *coid* (a file descriptor) from *open()* or **`ConnectAttach(..., _NTO_SIDE_CHANNEL, ...)`**.

For more information, see the section on “Channel IDs vs process IDs” in the Programming Issues chapter in this guide.

Sendfd()

QNX Neutrino equivalent:

MsgSend()

In migration library?

No

☞ This function takes a file descriptor (which is also a connection ID in QNX Neutrino).

For more information, see the section on “Channel IDs vs process IDs” in the Programming Issues chapter in this guide.

Sendfdmx()

QNX Neutrino equivalent:

MsgSendv()

In migration library?

No

☞ This function takes a file descriptor (which is also a connection ID in QNX Neutrino).

For more information, see the section on "Channel IDs vs process IDs" in the Programming Issues chapter in this guide.

Sendmx()

QNX Neutrino equivalent:

MsgSendv()

In migration library?

Yes

☞ This function takes a connection ID (*coid*) instead of a process ID. You can get this *coid* (a file descriptor) from *open()* or **`ConnectAttach(..., _NTO_SIDE_CHANNEL, ...)`**.

For more information, see the section on "Channel IDs vs process IDs" in the Programming Issues chapter in this guide.

_setmx()

QNX Neutrino equivalent:

SETIOV() for use with the QNX Neutrino *Msg*()* functions.

In migration library?

Yes — a *_setmx()* macro is provided in the migration library header file.

set_new_handler()

QNX Neutrino equivalent:

N/A

In migration library?

No

This is available in the C++ library (posted as free software for QNX Neutrino).

_set_new_handler()

QNX Neutrino equivalent:

N/A

In migration library?

No

This is available in the C++ library (posted as free software for QNX Neutrino).

sound()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_splitpath()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_splitpath2()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

stackavail()

QNX Neutrino equivalent:

__stackavail()

In migration library?

No

_status87()

QNX Neutrino equivalent:

No longer supported.

In migration library?

No

_strdate()

QNX Neutrino equivalent:

time(), *localtime()*, *gmtime()*, and *strftime()*

In migration library?

No

_strdup()

QNX Neutrino equivalent:

strdup()

In migration library?

No

_stricmp()

QNX Neutrino equivalent:

stricmp()

In migration library?

No

_strlwr()

QNX Neutrino equivalent:

strlwr()

In migration library?

No

_strnicmp()

QNX Neutrino equivalent:

strnicmp()

In migration library?

No

_strrev()

QNX Neutrino equivalent:

strrev()

In migration library?

No

_strtime()

QNX Neutrino equivalent:

time(), *localtime()*, *gmtime()*, and *strftime()*

In migration library?

No

_strupr()

QNX Neutrino equivalent:

strupr()

In migration library?

No

tcsetct()

QNX Neutrino equivalent:

N/A

In migration library?

No

Although there's no equivalent function for this, the first tty that is opened without O_NOCTTY after a call to *setsid()* and that doesn't already have a controlling process will cause the calling process to be the controlling process for this tty.

*term_** functions

The following functions have no QNX Neutrino equivalent, and aren't in the migration library; use **ncurses** instead.

term_attr_type()
term_axis()
term_bar()
term_box()
term_box_fill()
term_box_off()
term_box_on()
term_clear()
term_color()
term_cur()
term_delete_char()
term_delete_line()
term_down()
term_field()
term_fill()
term_flush()
term_get_line()
term_home()
term_init()
term_insert_char()
term_insert_line()
term_insert_off()
term_insert_on()
term_key()
term_left()
term_lmenu()
term_load()
term_menu()
term_mouse_default()
term_mouse_flags()
term_mouse_handler()
term_mouse_hide()
term_mouse_move()
term_mouse_off()
term_mouse_on()
term_mouse_process()
term_printf()
term_receive()
term_relearn_size()
term_resize_off()
term_resize_on()
term_restore()
term_restore_image()
term_right()
term_save_image()
term_scroll_down()

term_scroll_up()	*term_up()*
term_setup()	*term_video_off()*
term_type()	*term_video_on()*
term_unkey()	*term_window_scan()*

tfork()

QNX Neutrino equivalent:

N/A

In migration library?

No

QNX Neutrino has true POSIX threads. See the *pthread_*()* functions (specifically, *pthread_create()*) as a starting point.

timer_create()

QNX Neutrino equivalent:

timer_create()

In migration library?

No

The QNX 4 version was based on a draft standard. In QNX Neutrino, the timer ID is returned through the third parameter, and the **`sigevent`** structure is filled in differently.

_tolower()

QNX Neutrino equivalent:

tolower()

In migration library?

No

_toupper()

QNX Neutrino equivalent:

toupper()

In migration library?

No

Trace0()

The following functions have no QNX Neutrino equivalent, and aren't in the migration library; consider using *syslog()* for logging instead:

Trace0()	*Trace4()*
Trace0b()	*Trace4b()*
Trace1()	*Trace5()*
Trace2()	*Trace5b()*
Trace2b()	*Trace6()*
Trace3()	*Trace6b()*

Trigger()

QNX Neutrino equivalent:

N/A

In migration library?

Yes

Proxies have been replaced by *pulses*. See the section on "Proxies vs pulses" in the chapter on Programming Issues in this guide.

The migration library *Trigger()* function works the same as the QNX 4 one, but it's slower if the "triggerer" is in a different process than that which the proxy is attached to. For details, see the *Trigger()* function in the Migration Library chapter in this guide.

umount()

QNX Neutrino equivalent:

umount()

In migration library?

No

This is supported, but it now has unused flags parameters.

ungetch()

QNX Neutrino equivalent:

ungetc()

In migration library?

No

unlock()

QNX Neutrino equivalent:

fcntl() with F_SETLK

In migration library?

No

_vbprintf()

QNX Neutrino equivalent:

vsnprintf()

In migration library?

No

vcprintf()

QNX Neutrino equivalent:

vprintf(), with **`/dev/tty`** set as standard output.

In migration library?

No

vcscanf()

QNX Neutrino equivalent:

vsscanf(), with **`/dev/tty`** set as standard output.

In migration library?

No

Writemsg()

QNX Neutrino equivalent:

MsgWrite()

In migration library?

Yes

☞ This function takes the receive ID returned from *MsgReceive()* instead of a process ID.

Writemsgmx()

QNX Neutrino equivalent:

MsgWritev()

In migration library?

Yes

☞ This function takes the receive ID returned from *MsgReceive()* instead of a process ID.

Yield()

QNX Neutrino equivalent:

sched_yield()

In migration library?

Yes

Appendix B

QNX 4 functions supported by QNX Neutrino

Some functions from the QNX 4 C library are present in QNX Neutrino, but behave differently or have a slightly different set of arguments to meet the POSIX 1003.1 specification.

These QNX 4 functions are currently available to QNX Neutrino programs (note that most of them are part of the ANSI C library or POSIX 1003.1 spec):

abort()
abs()
accept()
access()
acos()
acosh()
alarm()
alloca()
asctime()
asin()
asinh()
assert()
atan()
atan2()
atanh()
atexit()
atof()
atoh()
atoi()
atol()

basename()
bcmp()
bcopy()
bind()
bindresvport()
brk()
bsearch()
bzero()

cabs()
calloc()
ceil()
cfgetispeed()
cfgetospeed()
cfree()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
chroot()
chsize()
clearenv()
clearerr()
clock()
clock_getres()
clock_gettime()
clock_nanosleep()
clock_settime()
close()
closedir()
closelog()
_cmdname()
confstr()
connect()
cos()
cosh()
creat()
ctermid()
ctime()

delay()
difftime()
div()
dn_comp()
dn_expand()
ds_clear()
ds_create()
ds_deregister()
ds_flags()
ds_get()
ds_register()
ds_set()
dup()
dup2()

eaccess()
endgrent()
endhostent()
endnetent()
endprotoent()

endpwent()
endservent()
environ
eof()
errno
execl()
execle()
execlp()
execlpe()
execv()
execve()
execvp()
execvpe()
_exit()
exit()
exp()

fabs()
fchmod()
fchown()
fclose()
fcloseall()
fcntl()
fdatasync()
fdopen()
feof()
ferror()
fflush()
ffs()
fgetc()
fgetchar()
fgetpos()
fgets()
fileno()
floor()
flushall()
fmod()
fnmatch()
fopen()
fork()
fpathconf()
fprintf()
fputc()
fputchar()
fputs()
fread()
free()
freopen()
frexp()
fscanf()
fseek()
fsetpos()
fstat()
fsync()
ftell()
ftime()
ftruncate()
ftw()
fwrite()

getc()
getchar()
getcwd()
getdtablesize()
getegid()
getenv()
geteuid()
getgid()
getgrent()
getgrgid()
getgrnam()
getgrouplist()
getgroups()
gethostbyaddr()
gethostbyname()
gethostent()
gethostname()
getitimer()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getpass()
getpeername()
getpgid()
getpgrp()
getpid()
getppid()
getprio()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()

getpwnam()
getpwuid()
getrusage()
gets()
getservbyname()
getservbyport()
getservent()
getsid()
getsockname()
getsockopt()
gettimeofday()
getuid()
getw()
gmtime()

h_errno
herror()
hostent
hstrerror()
htonl()
htons()
hypot()

index()
inet_addr()
inet_aton()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()
inet_ntop()
inet_pton()
input_line()
ioctl()
isalnum()
isalpha()
isascii()
isatty()
iscntrl()
isdigit()
isfdtype()
isgraph()
islower()
isprint()
ispunct()
isspace()
isupper()
isxdigit()
itoa()

j0()
j1()
jn()

kill()
killpg()

labs()
ldexp()
ldiv()
link()
listen()
localeconv()
localtime()
log()
log10()
login_tty()
longjmp()
lseek()
lstat()
ltoa()
ltrunc()

main()
malloc()
max()
mblen()
mbstowcs()
mbtowc()
memccpy()
memchr()
memcmp()
memcpy()
memicmp()
memmove()
memset()
min()
mkdir()
mkfifo()
mknod()
mktemp()
mktime()
mmap()
modem_open()

modem_read()
modem_script()
modem_write()
modf()
mprotect()
mq_close()
mq_getattr()
mq_notify()
mq_open()
mq_receive()
mq_send()
mq_setattr()
mq_unlink()
munmap()

nanosleep()
`netent`
nice()
ntohl()
ntohs()

offsetof()
open()
opendir()
openlog()

pathconf()
pause()
pclose()
perror()
pipe()
popen()
pow()
printf()
`protoent`
putc()
putchar()
putenv()
puts()
putw()

qsort()

Raccept()
raise()
rand()
random()
Rbind()
rcmd()
Rconnect()
rdchk()
read()
readdir()
readlink()
readv()
realloc()
realpath()
re_comp()
recv()
recvfrom()
recvmsg()
re_exec()
regcomp()
regerror()
regexec()
regfree()
remove()
rename()
res_init()
res_mkquery()
res_query()
res_querydomain()
res_search()
res_send()
rewind()
rewinddir()
Rgetsockname()
rindex()
Rlisten()
rmdir()
Rrcmd()
rresvport()
Rselect()
ruserok()

sbrk()
scandir()
scanf()
sched_getparam()
sched_getscheduler()
sched_setparam()
sched_setscheduler()
sched_yield()
searchenv()
select()
sem_destroy()

sem_init()
sem_post()
sem_trywait()
sem_wait()
send()
sendmsg()
sendto()
servent
setbuf()
setegid()
setenv()
seteuid()
setgid()
setgrent()
sethostent()
sethostname()
setitimer()
setjmp()
setlinebuf()
setlocale()
setlogmask()
setnetent()
setpgid()
setpgrp()
setprio()
setprotoent()
setpwent()
setregid()
setreuid()
setservent()
setsid()
setsockopt()
settimeofday()
setuid()
setvbuf()
shm_open()
shm_unlink()
shutdown()
sigaction()
sigaddset()
sigblock()
sigdelset()
sigemptyset()
sigevent
sigfillset()
sigismember()
siglongjmp()
sigmask()
signal()
sigpending()
sigprocmask()
sigsetjmp()
sigsuspend()
sin()
sinh()
sleep()
snprintf()
sockatmark()
socket()
SOCKSinit()
sopen()
spawnl()
spawnle()
spawnlp()
spawnlpe()
spawnv()
spawnve()
spawnvp()
spawnvpe()
sprintf()
sqrt()
srand()
sscanf()
stat()
strcasecmp()
strcat()
strchr()
strcmp()
strcmpi()
strcoll()
strcpy()
strcspn()
strdup()
strerror()
strftime()
stricmp()
strlen()
strlwr()
strncat()
strncmp()
strncpy()
strnicmp()

strnset()
strpbrk()
strrchr()
strrev()
strsep()
strset()
strsignal()
strspn()
strstr()
strtod()
strtok()
strtol()
strtoul()
strupr()
strxfrm()
symlink()
sync()
sysconf()
syslog()
system()

tan()
tanh()
tcdrain()
tcdropline()
tcflow()
tcflush()
tcgetattr()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()
tell()
tempnam()
time()
timer_delete()
timer_gettime()
timer_settime()
times()
tmpfile()
tmpnam()
tolower()
toupper()
truncate()
ttyname()
tzset()

ultoa()
umask()
uname()
ungetc()
unlink()
usleep()
utime()
utimes()
utoa()

va_arg()
va_end()
va_start()
vfork()
vfprintf()
vfscanf()
vprintf()
vsprintf()
vsscanf()
vsyslog()

wait()
waitpid()
wcstombs()
wctomb()
write()

y0()
y1()
yn()

These QNX 4 functions are available under QNX Neutrino, but have a different API or usage:

- *getwd()*
- *glob()*
- *inet_ntoa_r()*
- *timer_create()* — the QNX 4 version was based on a draft standard.

- *writev()*
- *crypt()* — a Unix-compatible version; for the QNX 4 version, use *qnx_crypt()*