QNX® Neutrino® Realtime Operating System

Programmer's Guide

For QNX® Neutrino® 6.4

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/`

Electronic edition published 2009

Contents

List of Figures

About This Book

What you'll find in this guide

The Neutrino *Programmer's Guide* is intended for developers who are building applications that will run under the QNX Neutrino Realtime Operating System.

Depending on the nature of your application and target platform, you may also need to refer to *Building Embedded Systems*. If you're using the Integrated Development Environment, see the IDE *User's Guide*. For a different perspective on programming in Neutrino, see *Getting Started with QNX Neutrino: A Guide for Realtime Programmers*.

This table may help you find what you need in the *Programmer's Guide*:

When you want to:	**Go to:**
Get started with a "Hello, world!" program	Compiling and Debugging
Get an overview of the Neutrino process model and scheduling methods	Programming Overview
Create and terminate processes	Processes
Understand the inaccuracies in times	Tick, Tock: Understanding the Neutrino Microkernel's Concept of Time
Use native networking	Transparent Distributed Processing Using Qnet
Learn about ISRs in Neutrino	Writing an Interrupt Handler
Analyze and detect problems related to dynamic memory management	Heap Analysis: Making Memory Errors a Thing of the Past
Deal with non-x86 issues (e.g. big-endian vs little-endian)	Freedom from Hardware and Platform Dependencies
Understand our **`Makefile`** methodology	Conventions for Recursive Makefiles and Directories
Learn how to use the GDB debugger	Using GDB
Find out about using memory on ARM targets	ARM Memory Management
Find out about advanced Qnet topics	Advanced Qnet Topics
Look up terms used in the Neutrino documentation	Glossary

We assume that you've already installed Neutrino and that you're familiar with its architecture. For a detailed overview, see the *System Architecture* manual.

Recommended reading

For the most part, the information that's documented in the *Programmer's Guide* is specific to QNX. For more general information, we recommend the following books:

Threads:

- Butenhof, David R. 1997. *Programming with POSIX Threads*. Reading, MA: Addison-Wesley Publishing Company. ISBN 0-201-63392-2.

TCP/IP programming (note that some of the advanced API features mentioned in the following books might not be supported):

- Hunt, Craig. 2002. *TCP/IP Network Administration*. Sebastopol, CA: O'Reilly & Associates. ISBN 0-596-00297-1.
- Stevens, W. Richard. 1997. *Unix Network Programming: Networking APIs: Sockets and XTI*. Upper Saddle River, NJ: Prentice-Hall PTR. ISBN 0-13-490012-X.
- —. 1993. *TCP/IP Illustrated, Volume 1 The Protocols*. Reading, MA: Addison-Wesley Publishing Company. ISBN 0-201-63346-9.
- —. 1995. *TCP/IP Illustrated, Volume 2 The Implementation*. Reading, MA: Addison-Wesley Publishing Company. ISBN 0-201-63354-X.

Typographical conventions

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

Reference	Example
Code examples	`if( stream == NULL )`
Command options	**`-lR`**
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	**`/dev/null`**
Function names	*exit()*

continued...

Reference	Example
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	`something you type`
Keyboard keys	Enter
Program output	`login:`
Programming constants	NULL
Programming data types	`unsigned short`
Programming literals	`0xFF`, `"message string"`
Variable names	*stdin*
User-interface components	**Cancel**

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

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

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

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

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

Note to Windows users

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

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

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

Chapter 1

Compiling and Debugging

In this chapter. . .

Choosing the version of the OS

The QNX Momentics Tool Suite lets you install and work with multiple versions of Neutrino. Whether you're using the command line or the IDE, you can choose which version of the OS to build programs for.

When you install QNX Momentics, you get a set of configuration files that indicate where you've install the software. The **QNX_CONFIGURATION** environment variable stores the location of the configuration files for the installed versions of Neutrino; on a self-hosted Neutrino machine, the default is `/etc/qconfig`.

If you're using the command-line tools, use the `qconfig` utility to configure your machine to use a specific version of the QNX Momentics Tool Suite.

On Windows hosts, use `QWinCfg`, a graphical front end for `qconfig`. You can launch it from the Start menu.

Here's what `qconfig` does:

- If you run it without any options, `qconfig` lists the versions that are installed on your machine.
- If you use the `-e` option, you can use `qconfig` to set up the environment for building software for a specific version of the OS. For example, if you're using the Korn shell (`ksh`), you can configure your machine like this:

  ```
  eval ‘qconfig -n "QNX Neutrino 6.3.0" -e‘
  ```

When you start the IDE, it uses your current `qconfig` choice as the default version of the OS; if you haven't chosen a version, the IDE chooses an entry from the directory identified by **QNX_CONFIGURATION**. If you want to override the IDE's choice, you can choose the appropriate build target. For details, see "Version coexistence" in the Concepts chapter of the IDE *User's Guide*.

Neutrino uses these environment variables to locate files on the *host* machine:

QNX_HOST	The location of host-specific files.
QNX_TARGET	The location of target backends on the host machine.

The `qconfig` utility sets these variables according to the version of QNX Momentics that you specified.

Making your code more portable

To help you create portable applications, QNX Neutrino lets you compile for specific standards and include QNX- or Neutrino-specific code.

Conforming to standards

The header files supplied with the C library provide the proper declarations for the functions and for the number and types of arguments used with them. Constant values used in conjunction with the functions are also declared. The files can usually be included in any order, although individual function descriptions show the preferred order for specific headers.

When you use the **-ansi** option, **qcc** compiles strict ANSI code. Use this option when you're creating an application that must conform to the ANSI standard. The effect on the inclusion of ANSI- and POSIX-defined header files is that certain portions of the header files are omitted:

- for ANSI header files, these are the portions that go beyond the ANSI standard
- for POSIX header files, these are the portions that go beyond the POSIX standard

You can then use the **qcc -D** option to define *feature-test macros* to select those portions that are omitted. Here are the most commonly used feature-test macros:

_POSIX_C_SOURCE=199506
: Include those portions of the header files that relate to the POSIX standard (*IEEE Standard Portable Operating System Interface for Computer Environments - POSIX 1003.1*, 1996)

_FILE_OFFSET_BITS=64
: Make the libraries use 64-bit file offsets.

_LARGEFILE64_SOURCE
: Include declarations for the functions that support large files (those whose names end with **64**).

_QNX_SOURCE
: Include everything defined in the header files. This is the default.

Feature-test macros may be defined on the command line, or in the source file before any header files are included. The latter is illustrated in the following example, in which an ANSI- and POSIX-conforming application is being developed.

```
#define _POSIX_C_SOURCE=199506
#include <limits.h>
#include <stdio.h>
   ⋮
#if defined(_QNX_SOURCE)
  #include "non_POSIX_header1.h"
  #include "non_POSIX_header2.h"
  #include "non_POSIX_header3.h"
#endif
```

You'd then compile the source code using the **-ansi** option.

The following ANSI header files are affected by the _POSIX_C_SOURCE feature-test macro:

- `<limits.h>`
- `<setjmp.h>`
- `<signal.h>`
- `<stdio.h>`
- `<stdlib.h>`
- `<time.h>`

The following ANSI and POSIX header files are affected by the _QNX_SOURCE feature-test macro:

Header file	Type
`<ctype.h>`	ANSI
`<fcntl.h>`	POSIX
`<float.h>`	ANSI
`<limits.h>`	ANSI
`<math.h>`	ANSI
`<process.h>`	extension to POSIX
`<setjmp.h>`	ANSI
`<signal.h>`	ANSI
`<sys/stat.h>`	POSIX
`<stdio.h>`	ANSI
`<stdlib.h>`	ANSI
`<string.h>`	ANSI
`<termios.h>`	POSIX
`<time.h>`	ANSI
`<sys/types.h>`	POSIX
`<unistd.h>`	POSIX

Including QNX- or Neutrino-specific code

If you need to include QNX- Neutrino-specific code in your application, you can wrap it in an **#ifdef** to make the program more portable. The **qcc** utility defines these preprocessor symbols (or *manifest constants*):

__QNX__ The target is a QNX operating system (QNX 4 or QNX Neutrino).

__QNXNTO__ The target is the QNX Neutrino operating system.

For example:

```
#if defined(__QNX__)
   /* QNX-specific (any flavor) code here */

   #if defined(__QNXNTO__)
      /* QNX Neutrino-specific code here */
   #else
      /* QNX 4-specific code here */
   #endif
#endif
```

For information about other preprocessor symbols that you might find useful, see the Manifests chapter of the Neutrino *Library Reference*.

Header files in /usr/include

The ${**QNX_TARGET**}**/usr/include** directory includes at least the following subdirectories (in addition to the usual **sys**):

arpa ARPA header files concerning the Internet, FTP and TELNET.

hw Descriptions of various hardware devices.

arm, **mips**, **ppc**, **sh**, **x86**

CPU-specific header files. You typically don't need to include them directly — they're included automatically. There are some files that you might want to look at:

- Files ending in ***intr.h** describe interrupt vector numbers for use with *InterruptAttach()* and *InterruptAttachEvent()*.
- Files ending with ***cpu.h** describe the registers and other information about the processor.

malloc, **malloc_g**

Memory allocation; for more information, see the Heap Analysis: Making Memory Errors a Thing of the Past chapter in this guide.

net Network interface descriptions.

netinet, **netinet6**, **netkey**

Header files concerning TCP/IP.

photon Header files concerning the Photon microGUI; for more information, see the Photon documentation.

snmp Descriptions for the Simple Network Management Protocol (SNMP).

Self-hosted or cross-development

In the rest of this chapter, we'll describe how to compile and debug a Neutrino system. Your Neutrino system might be anything from a deeply embedded turnkey system to a powerful multiprocessor server. You'll develop the code to implement your system using development tools running on the Neutrino platform itself or on any other supported cross-development platform.

Neutrino supports both of these development types:

- *self-hosted* — you develop and debug on the same x86 system that's running the QNX Neutrino OS
- *cross-development* — you develop on your host system, then transfer and debug the executable on your target hardware

This section describes the procedures for compiling and debugging for both types.

A simple example

We'll now go through the steps necessary to build a simple Neutrino system that runs on a standard PC and prints out the text "Hello, world!" — the classic first C program.

Let's look at the spectrum of methods available to you to run your executable:

If your environment is:	**Then you can:**
Self-hosted	Compile and link, then run on host
Cross-development, network filesystem link	Compile and link, load over network filesystem, then run on target
Cross-development, debugger link	Compile and link, use debugger as a "network filesystem" to transfer executable over to target, then run on target
Cross-development, rebuilding the image	Compile and link, rebuild entire image, reboot target.

Which method you use depends on what's available to you. All the methods share the same initial step — write the code, then compile and link it for Neutrino on the platform that you wish to run the program on.

You can choose how you wish to compile and link your programs: you can use tools with a command-line interface (via the **qcc** command) or you can use an IDE (Integrated Development Environment) with a graphical user interface (GUI) environment. Our samples here illustrate the command-line method.

The “Hello, world!” program itself is very simple:

```
#include <stdio.h>

int
main (void)
{
    printf ("Hello, world!\n");
    return (0);
}
```

You compile it for PowerPC (big-endian) with the single line:

```
qcc -V gcc_ntoppcbe hello.c -o hello
```

This executes the C compiler with a special cross-compilation flag, **-V gcc_ntoppcbe**, that tells the compiler to use the **gcc** compiler, Neutrino-specific includes, libraries, and options to create a PowerPC (big-endian) executable using the GCC compiler.

To see a list of compilers and platforms supported, simply execute the command:

```
qcc -V
```

If you’re using an IDE, refer to the documentation that came with the IDE software for more information.

At this point, you should have an executable called **hello**.

Self-hosted

If you’re using a self-hosted development system, you’re done. You don’t even have to use the **-V** cross-compilation flag (as was shown above), because the **qcc** driver will default to the current platform. You can now run **hello** from the command line:

```
hello
```

Cross-development with network filesystem

If you’re using a network filesystem, let’s assume you’ve already set up the filesystem on both ends. For information on setting this up, see the Sample Buildfiles appendix in *Building Embedded Systems*.

Using a network filesystem is the richest cross-development method possible, because you have access to remotely mounted filesystems. This is ideal for a number of reasons:

- Your embedded system requires only a network connection; no disks (and disk controllers) are required.
- You can access all the shipped and custom-developed Neutrino utilities — they don’t need to be present on your (limited) embedded system.

- Multiple developers can share the same filesystem server.

For a network filesystem, you'll need to ensure that the shell's **PATH** environment variable includes the path to your executable via the network-mounted filesystem. At this point, you can just type the name of the executable at the target's command-line prompt (if you're running a shell on the target):

```
hello
```

Cross-development with debugger

Once the debug agent is running, and you've established connectivity between the host and the target, you can use the debugger to download the executable to the target, and then run and interact with it.

Download/upload facility

When the debug agent is connected to the host debugger, you can transfer files between the host and target systems. Note that this is a general-purpose file transfer facility — it's not limited to transferring only executables to the target (although that's what we'll be describing here).

In order for Neutrino to execute a program on the target, the program must be available for loading from some type of filesystem. This means that when you transfer executables to the target, you must write them to a filesystem. Even if you don't have a conventional filesystem on your target, recall that there's a writable "filesystem" present under Neutrino — the **/dev/shmem** filesystem. This serves as a convenient RAM-disk for downloading the executables to.

Cross-development, deeply embedded

If your system is deeply embedded and you have no connectivity to the host system, or you wish to build a system "from scratch," you'll have to perform the following steps (in addition to the common step of creating the executable(s), as described above):

1. Build a Neutrino system image.
2. Transfer the system image to the target.
3. Boot the target.

Step 1: Build a Neutrino system image.

You use a *buildfile* to build a Neutrino system image that includes your program. The buildfile contains a list of files (or modules) to be included in the image, as well as information about the image. A buildfile lets you execute commands, specify command arguments, set environment variables, and so on. The buildfile will look like this:

```
[virtual=ppcbe,elf] .bootstrap = {
    startup-800fads
```

```
    PATH=/proc/boot procnto-800
}
[+script] .script = {
    devc-serppc800 -e -c20000000 -b9600 smc1 &
    reopen
    hello
}

[type=link] /dev/console=/dev/ser1
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[perms=+r,+x]
libc.so

[data=copy]
[perms=+r,+x]
devc-serppc800
hello &
```

The first part (the four lines starting with **[virtual=ppcbe,elf]**), contains information about the kind of image we're building.

The next part (the five lines starting with **[+script]**) is the startup script that indicates what executables (and their command-line parameters, if any) should be invoked.

The **[type=link]** lines set up symbolic links to specify the serial port and shared library file we want to use.

The runtime linker is expected to be found in a file called **ldqnx.so.2**, but the runtime linker is currently contained within the **libc.so** file, so we make a process manager symbolic link to it.

The **[perms=+r,+x]** lines assign permissions to the binaries that follow — in this case, we're setting them to be Readable and Executable.

Then we include the C shared library, **libc.so**.

Then the line **[data=copy]** specifies to the loader that the data segment should be copied. This applies to all programs that follow the **[data=copy]** attribute. The result is that we can run the executable multiple times.

Finally, the last part (the last two lines) is simply the list of files indicating which files should be included as part of the image. For more details on buildfile syntax, see the **mkifs** entry in the *Utilities* Reference.

Our sample buildfile indicates the following:

- A PowerPC 800 FADS board and ELF boot prefix code are being used to boot.
- The image should contain **devc-serppc800**, the serial communications manager for the PowerPC 80x family, as well as **hello** (our test program).
- **devc-serppc800** should be started in the background (specified by the **&** character). This manager will use a clock rate of 20 MHz, a baud rate of 9600, and an **smc1** device.

- Standard input, output, and error should be redirected to **`/dev/ser1`** (via the **`reopen`** command, which by default redirects to **`/dev/console`**, which we've linked to **`/dev/ser1`**).
- Finally, our **`hello`** program should run.

Let's assume that the above buildfile is called **`hello.bld`**. Using the **`mkifs`** utility, you could then build an image by typing:

```
mkifs hello.bld hello.ifs
```

Step 2: Transfer the system image to the target.

You now have to transfer the image **`hello.ifs`** to the target system. If your target is a PC, the most universal method of booting is to make a bootable floppy diskette.

If you're developing on a platform that has TCP/IP networking and connectivity to your target, you may be able to boot your Neutrino target system using a BOOTP server. For details, see the "BOOTP section" in the Customizing IPL Programs chapter in *Building Embedded Systems*.

If your development system is Neutrino, transfer your image to a floppy by issuing this command:

```
dinit -f hello.ifs /dev/fd0
```

If your development system is Windows NT or Windows 95/98, transfer your image to a floppy by issuing this command:

```
dinit -f hello.ifs a:
```

Step 3: Boot the target.

Place the floppy diskette into your target system and reboot your machine. The message "**`Hello, world!`**" should appear on your screen.

Using libraries

When you're developing code, you almost always make use of a *library* — a collection of code modules that you or someone else has already developed (and hopefully debugged). Under Neutrino, we have three different ways of using libraries:

- static linking
- dynamic linking
- runtime loading

Static linking

You can combine your modules with the modules from the library to form a single executable that's entirely self-contained. We call this *static linking*. The word "static" implies that it's not going to change — *all* the required modules are already combined into one executable.

Dynamic linking

Rather than build a self-contained executable ahead of time, you can take your modules and link them in such a way that the Process Manager will link them to the library modules before your program runs. We call this *dynamic linking*. The word "dynamic" here means that the association between your program and the library modules that it uses is done *at load time*, not at link time (as was the case with the static version).

Runtime loading

There's a variation on the theme of dynamic linking called *runtime loading*. In this case, the program decides *while it's actually running* that it wishes to load a particular function from a library.

Static and dynamic libraries

To support the two major kinds of linking described above, Neutrino has two kinds of libraries: *static* and *dynamic*.

Static libraries

A static library is usually identified by a **.a** (for "archive") suffix (e.g. **libc.a**). The library contains the modules you want to include in your program and is formatted as a collection of ELF object modules that the linker can then extract (as required by your program) and *bind* with your program at link time.

This "binding" operation literally copies the object module from the library and incorporates it into your "finished" executable. The major advantage of this approach is that when the executable is created, it's entirely self-sufficient — it doesn't require any other object modules to be present on the target system. This advantage is usually outweighed by two principal disadvantages, however:

- *Every* executable created in this manner has its own private copy of the library's object modules, resulting in large executable sizes (and possibly slower loading times, depending on the medium).
- You must *relink the executable* in order to upgrade the library modules that it's using.

Dynamic libraries

A dynamic library is usually identified by a **.so** (for "shared object") suffix (e.g. **libc.so**). Like a static library, this kind of library also contains the modules that you want to include in your program, but these modules are *not* bound to your program at link time. Instead, your program is linked in such a way that the Process Manager causes your program to be bound to the shared objects at load time.

The Process Manager performs this binding by looking at the program to see if it references any shared objects (**.so** files). If it does, then the Process Manager looks to see if those particular shared objects are already present in memory. If they're not, it loads them into memory. Then the Process Manager patches your program to be able to use the shared objects. Finally, the Process Manager starts your program.

Note that from your program's perspective, it isn't even aware that it's running with a shared object versus being statically linked — that happened before the first line of your program ran!

The main advantage of dynamic linking is that the programs in the system will reference only a particular set of objects — they don't contain them. As a result, programs are smaller. This also means that you can upgrade the shared objects *without relinking the programs*. This is especially handy when you don't have access to the source code for some of the programs.

dlopen()

When a program decides at runtime that it wants to "augment" itself with additional code, it will issue the *dlopen()* function call. This function call tells the system that it should find the shared object referenced by the *dlopen()* function and create a binding between the program and the shared object. Again, if the shared object isn't present in memory already, the system will load it. The main advantage of this approach is that the program can determine, at runtime, which objects it needs to have access to.

Note that there's no *real* difference between a library of shared objects that you link against and a library of shared objects that you load at runtime. Both modules are of the exact same format. The only difference is in how they get used.

By convention, therefore, we place libraries that you link against (whether statically or dynamically) into the **lib** directory, and shared objects that you load at runtime into the **lib/dll** (for "dynamically loaded libraries") directory.

Note that this is just a convention — there's nothing stopping you from linking against a shared object in the **lib/dll** directory or from using the *dlopen()* function call on a shared object in the **lib** directory.

Platform-specific library locations

The development tools have been designed to work out of their processor directories (**x86**, **ppcbe**, etc.). This means you can use the same toolset for any target platform.

If you have development libraries for a certain platform, then put them into the platform-specific library directory (e.g. **/x86/lib**), which is where the compiler tools will look.

You can use the **-L** option to **qcc** to explicitly provide a library path.

Linking your modules

To link your application against a library, use the **-l** option to **qcc**, omitting the **lib** prefix and any extension from the library's name. For example, to link against **libsocket**, specify **-l socket**.

You can specify more than one **-l** option. The **qcc** configuration files might specify some libraries for you; for example, **qcc** usually links against **libc**. The description of each function in the Neutrino *Library Reference* tells you which library to link against.

By default, the tool chain links dynamically. We do this because of all the benefits mentioned above.

If you want to link statically, then you should specify the **-static** option to **qcc**, which will cause the link stage to look in the library directory *only* for static libraries (identified by a **.a** extension).

For this release of Neutrino, you can't use the floating point emulator (**fpemu.so**) in statically linked executables.

Although we generally discourage linking statically, it does have this advantage: in an environment with tight configuration management and software QA, the very same executable can be regenerated at link time and known to be complete at runtime.

To link dynamically (the default), you don't have to do anything.

To link statically *and* dynamically (some libraries linked one way, other libraries linked the other way), the two keywords **-Bstatic** and **-Bdynamic** are positional parameters that can be specified to **qcc**. All libraries specified after the particular **-B** option will be linked in the specified manner. You can have multiple **-B** options:

```
qcc ... -Bdynamic -l1 -l2 -Bstatic -l3 -l4 -Bdynamic -l5
```

This will cause libraries **lib1**, **lib2**, and **lib5** to be dynamically linked (i.e. will link against the files **lib1.so**, **lib2.so** and **lib5.so**), and libraries **lib3** and **lib4** to be statically linked (i.e. will link against the files **lib3.a** and **lib4.a**).

You may see the extension **.1** appended to the name of the shared object (e.g. **libc.so.1**). This is a version number. Use the extension **.1** for your first revision, and increment the revision number if required.

You may wish to use the above "mixed-mode" linking because some of the libraries you're using will be needed by only one executable or because the libraries are small (less than 4 KB), in which case you'd be wasting memory to use them as shared libraries. Note that shared libraries are typically mapped in 4-KB pages and will require at least one page for the "**`text`**" section and possibly one page for the "**`data`**" section.

> When you specify **`-Bstatic`** or **`-Bdynamic`**, *all* subsequent libraries will be linked in the specified manner.

Creating shared objects

To create a shared object suitable for linking against:

1. Compile the source files for the library using the **`-shared`** option to **`qcc`**.
2. To create the library from the individual object modules, simply combine them with the linker (this is done via the **`qcc`** compiler driver as well, also using the **`-shared`** command-line option).

> Make sure that all objects and "static" libs that are pulled into a **`.so`** are position-independent as well (i.e. also compiled with **`-shared`**).

If you make a shared library that has to static-link against an existing library, you can't static-link against the **`.a`** version (because those libraries themselves aren't compiled in a position-independent manner). Instead, there's a special version of the libraries that has a capital "S" just before the **`.a`** extension. For example, instead of linking against **`libsocket.a`**, you'd link against **`libsocketS.a`**. We recommend that you don't static-link, but rather link against the **`.so`** shared object version.

Specifying an internal name

When you're building a shared object, you can specify the following option to **`qcc`**:

`"-Wl,-h`*name***`"`**

(You might need the quotes to pass the option through to the linker intact, depending on the shell.)

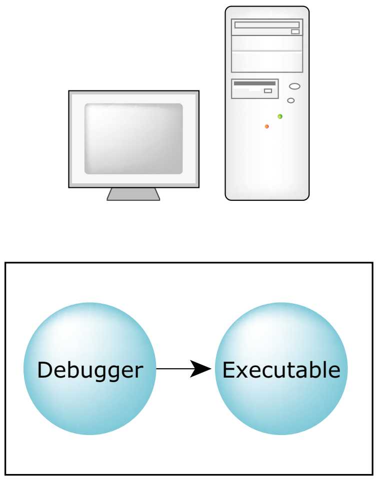
This option sets the internal name of the shared object to *name* instead of to the object's pathname, so you'd use *name* to access the object when dynamically linking. You might find this useful when doing cross-development (e.g. from a Windows NT system to a Neutrino target).

Debugging

Now let's look at the different options you have for debugging the executable. Just as you have two basic ways of developing (self-hosted and cross-development), you have similar options for debugging.

Debugging in a self-hosted environment

The debugger can run on the same platform as the executable being debugged:



Debugging in a self-hosted environment.

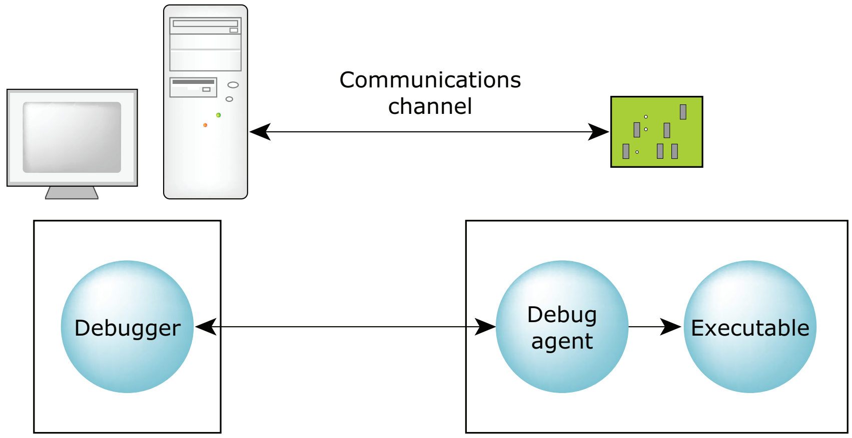
In this case, the debugger communicates directly with the program you're debugging. You can choose this type of debugging by running the `target procfs` command in the debugger — or by not running the `target` command at all.

It's also possible to use the `target qnx` command so that the debugger communicates with a local program via a debug agent, but this is the same as debugging in a cross-development environment.

A `procfs` session is possible only when the debugger and the program are on the same QNX Neutrino system.

Debugging in a cross-development environment

The debugger can run on one platform to debug executables on another:



Debugging in a cross-development environment.

In a cross-development environment, the host and the target systems must be connected via some form of communications channel.

The two components, the debugger and the debug agent, perform different functions. The debugger is responsible for presenting a user interface and for communicating over some communications channel to the debug agent. The debug agent is responsible for controlling (via the **/proc** filesystem) the process being debugged.

All debug information and source remains on the host system. This combination of a small target agent and a full-featured host debugger allows for full symbolic debugging, even in the memory-constrained environments of small targets.

In order to debug your programs with full source using the symbolic debugger, you'll need to tell the C compiler and linker to include symbolic information in the object and executable files. For details, see the **qcc** docs in the *Utilities* Reference. Without this symbolic information, the debugger can provide only assembly-language-level debugging.

The GNU debugger (gdb)

The GNU debugger is a command-line program that provides a very rich set of options. You'll find a tutorial-style doc called "Using GDB" as an appendix in this manual.

Starting gdb

You can invoke **gdb** by using the following variants, which correspond to your target platform:

For this target:	Use this command:
ARM	**ntoarm-gdb**
Intel	**ntox86-gdb**
MIPS	**ntomips-gdb**
PowerPC	**ntoppc-gdb**
SH4	**ntosh-gdb**

For more information, see the **gdb** entry in the *Utilities* Reference.

The process-level debug agent

When a breakpoint is encountered and the process-level debug agent (**pdebug**) is in control, the process being debugged and all its threads are stopped. All other processes continue to run and interrupts remain enabled.

To use the **pdebug** agent, you must set up pty support (via **devc-pty**) on your target.

When the process's threads are stopped and the debugger is in control, you may examine the state of any thread within the process. For more info on examining thread states, see your debugger docs.

The **pdebug** agent may either be included in the image and started in the image startup script or started later from any available filesystem that contains **pdebug**. The **pdebug** command-line invocation specifies which device will be used.

You can start **pdebug** in one of three ways, reflecting the nature of the connection between the debugger and the debug agent:

- serial connection
- TCP/IP static port connection
- TCP/IP dynamic port connection

Serial connection

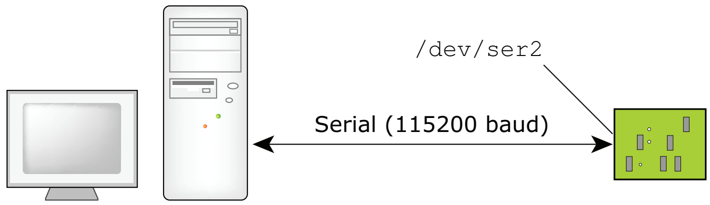
If the host and target systems are connected via a serial port, then the debug agent (**pdebug**) should be started with the following command:

pdebug *devicename*[,*baud*]

This indicates the target's communications channel (*devicename*) and specifies the baud rate (*baud*).

For example, if the target has a **/dev/ser2** connection to the host, and we want the link to be 115,200 baud, we would specify:

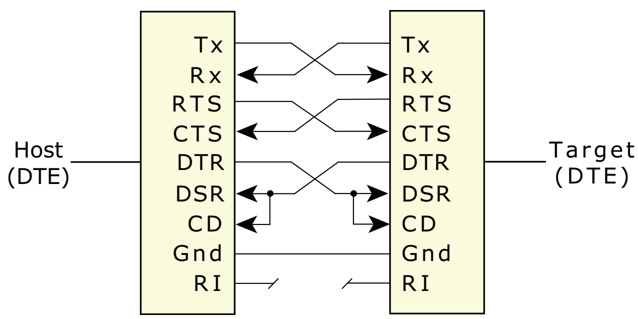
```
pdebug /dev/ser2,115200
```



Running the process debug agent with a serial link at 115200 baud.

The Neutrino target requires a supported serial port. The target is connected to the host using either a null-modem cable, which allows two identical serial ports to be directly connected, or a straight-through cable, depending on the particular serial port provided on the target.

The null-modem cable crosses the **`Tx/Rx`** data and handshaking lines. In our PowerPC FADS example, you'd use a a straight-through cable. Most computer stores stock both types of cables.



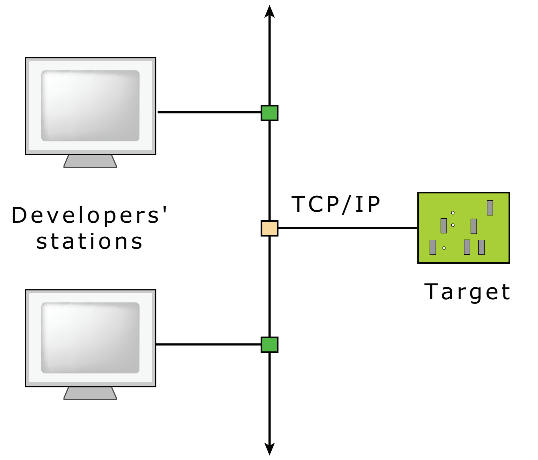
Null-modem cable pinout.

TCP/IP connection

If the host and the target are connected via some form of TCP/IP connection, the debugger and agent can use that connection as well. Two types of TCP/IP communications are possible with the debugger and agent: static port and dynamic port connections (see below).

The Neutrino target must have a supported Ethernet controller. Note that since the debug agent requires the TCP/IP manager to be running on the target, this requires more memory.

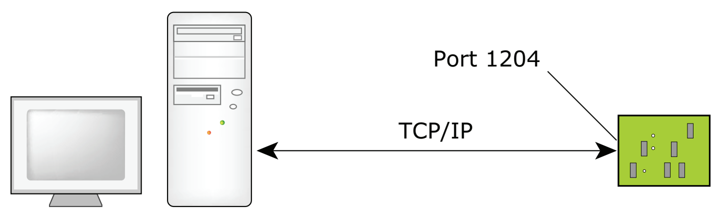
This need for extra memory is offset by the advantage of being able to run multiple debuggers with multiple debug sessions over the single network cable. In a networked development environment, developers on different network hosts could independently debug programs on a single common target.



Several developers can debug a single target system.

TCP/IP static port connection

For a static port connection, the debug agent is assigned a TCP/IP port number and will listen for communications on that port only. For example, the **pdebug 1204** command specifies TCP/IP port 1204:



Running the process debug agent with a TCP/IP static port.

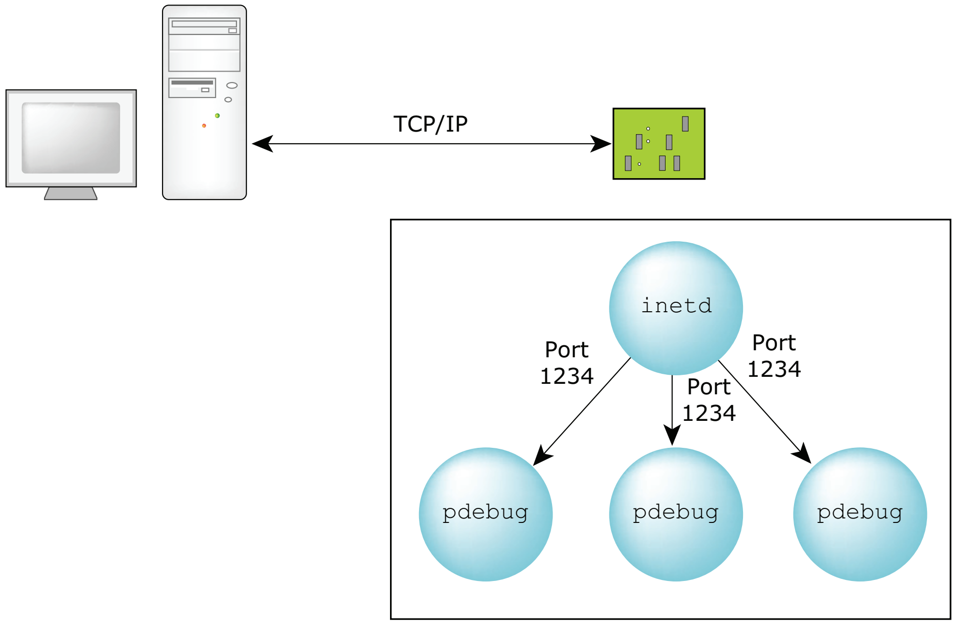
If you have multiple developers, each developer could be assigned a specific TCP/IP port number above the reserved ports `0` to `1024`.

TCP/IP dynamic port connection

For a dynamic port connection, the debug agent is started by **inetd** and communicates via standard input/output. The **inetd** process fetches the communications port from the configuration file (typically **/etc/services**). The host process debug agent connects to the port via **inetd** — the debug agent has no knowledge of the port.

The command to run the process debug agent in this case is simply as follows (from the **inetd.conf** file):

```
pdebug -
```



For a TCP/IP dynamic port connection, the **`inetd`** *process will manage the port.*

Note that this method is also suitable for one or more developers. It's effectively what the **`qconn`** daemon does to provide support to remote IDE components; **`qconn`** listens to a port and spawns **`pdebug`** on a new, dynamically determined port.

Sample buildfile for dynamic port sessions

The following buildfile supports multiple sessions specifying the same port. Although the port for each session on the **`pdebug`** side is the same, **`inetd`** causes unique ports to be used on the debugger side. This ensures a unique socket pair for each session.

Note that **`inetd`** should be included and started in your boot image. The **`pdebug`** program should also be in your boot image (or available from a mounted filesystem).

The config files could be built into your boot image (as in this sample buildfile) or linked in from a remote filesystem using the **`[type=link]`** command:

```
[type=link] /etc/services=/mount_point/services
[type=link] /etc/inetd.conf=/mount_point/inetd.conf
```

Here's the buildfile:

```
[virtual=x86,bios +compress] boot = {
    startup-bios -N node428
    PATH=/proc/boot:/bin:/apk/bin_nto:./ procnto
}

[+script] startup-script = {
# explicitly running in edited mode for the console link
    devc-ser8250 -e -b115200 &
    reopen
    display_msg Welcome to Neutrino on a PC-compatible BIOS system
# tcp/ip with a NE2000 Ethernet adaptor
```

```
    io-pkt-v4 -dne2000 -ptcpip if=ndi0:10.0.1.172 &
    waitfor /dev/socket
    inetd &
    pipe &
# pdebug needs devc-pty and esh
    devc-pty &
# NFS mount of the Neutrino filesystem
    fs-nfs3 -r 10.89:/x86 /x86 -r 10.89:/home /home &
# CIFS mount of the NT filesystem
    fs-cifs -b //QA:10.0.1.181:/QARoot /QAc apk 123 &
# NT Hyperterm needs this to interpret backspaces correctly
    stty erase=08
    reopen /dev/console
    [+session] esh &
}

[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[type=link] /lib=/x86/lib
[type=link] /tmp=/dev/shmem         # tmp points to shared memory
[type=link] /dev/console=/dev/ser2  # no local terminal
[type=link] /bin=/x86/bin           # executables in the path
[type=link] /apk=/home/apk          # home dir

[perms=+r,+x]           # Boot images made under MS-Windows
                        # need to be reminded of permissions.
devn-ne2000.so
libc.so
fpemu.so
libsocket.so

[data=copy]             # All executables that can be restarted
                        # go below.
devc-ser8250
io-pkt-v4
pipe
devc-pty
fs-nfs3
fs-cifs
inetd
esh
stty
ping
ls
                        # Data files are created in the named
                        # directory.
/etc/hosts = {
127.0.0.1    localhost
10.89        node89
10.222       node222
10.326       node326
10.0.1.181   QA node437
10.241       APP_ENG_1
}

/etc/services = {
ftp           21/tcp
telnet        23/tcp
finger        79/tcp
pdebug        8000/tcp
}

/etc/inetd.conf = {
ftp     stream    tcp    nowait    root    /bin/fdtpd      fdtpd
```

```
telnet  stream    tcp    nowait    root    /bin/telnetd    telnetd
finger  stream    tcp    nowait    root    /bin            fingerd
pdebug  stream    tcp    nowait    root    /bin/pdebug     pdebug -
}
```

A simple debug session

In this example, we'll be debugging our "Hello, world!" program via a TCP/IP link. We go through the following steps:

- configuring the target
- compiling for debugging
- starting the debug session
- getting help

Configure the target

Let's assume an x86 target using a basic TCP/IP configuration. The following lines (from the sample boot file at the end of this chapter) show what's needed to host the sample session:

```
io-pkt-v4 -dne2000 -ptcpip if=ndi0:10.0.1.172 &
devc-pty &
[+session] pdebug 8000 &
```

The above specifies that the host IP address is 10.0.1.172 (or 10.428 for short). The **pdebug** program is configured to use port 8000.

Compile for debugging

We'll be using the x86 compiler. Note the **-g** option, which enables debugging information to be included:

```
$ qcc -V gcc_ntox86 -g -o hello hello.c
```

Start the debug session

For this simple example, the sources can be found in our working directory. The **gdb** debugger provides its own shell; by default its prompt is **(gdb)**. The following commands would be used to start the session. To reduce document clutter, we'll run the debugger in quiet mode:

```
# Working from the source directory:
    (61) con1 /home/allan/src >ntox86-gdb -quiet

# Specifying the target IP address and the port
# used by pdebug:
    (gdb) target qnx 10.428:8000
    Remote debugging using 10.428:8000
```

```
    0x0 in ?? ()

# Uploading the debug executable to the target:
# (This can be a slow operation. If the executable
# is large, you may prefer to build the executable
# into your target image.)
# Note that the file has to be in the target system's namespace,
# so we can get the executable via a network filesystem, ftp,
# or, if no filesystem is present, via the upload command.

    (gdb) upload hello /tmp/hello

# Loading the symbolic debug information from the
# current working directory:
# (In this case, "hello" must reside on the host system.)

    (gdb) sym hello
    Reading symbols from hello...done.

# Starting the program:
    (gdb) run /tmp/hello
    Starting program:  /tmp/hello
    Trying to find symbol file for ldqnx.so.2
    Retrying dynamic interpreter in libc.so.1

# Setting the breakpoint on main():
    (gdb) break main
    Breakpoint 1 at 0x80483ae: file hello.c, line 8.

# Allowing the program to continue to the breakpoint
# found at main():
    (gdb) c
    Continuing.
    Breakpoint 1, main () at hello.c:8
    8       setprio (0,9);

# Ready to start the debug session.
(gdb)
```

Get help

While in a debug session, any of the following commands could be used as the next action for starting the actual debugging of the project:

n — Step through the program, proceeding through subroutine calls.

l — List the specified function or line.

break — Set a breakpoint on the specified function or line.

help — Get the help main menu.

help data — Get the help data menu.

help inspect — Get help for the **inspect** command.

inspect *y* — Inspect the contents of variable *y*.

`set y=3`	Assign a value to variable *y*.
`bt`	Get a back trace.

For more information about these commands and their arguments, see the Using GDB appendix in this guide, or use the **`help`** *cmd* command in **`gdb`**.

Let's see how to use some of these basic commands.

```
# The list command:
    (gdb) l
    3
    4   main () {
    5
    6       int x,y,z;
    7
    8       setprio (0,9);
    9       printf ("Hi ya!\n");
    10
    11      x=3;
    12      y=2;

# Press <enter> to repeat the last command:
    (gdb) <enter>
    13      z=3*2;
    14
    15      exit (0);
    16
    17  }

# Break on line 11:
   (gdb) break 11
   Breakpoint 2 at 0x80483c7: file hello.c, line 11.

# Continue until the first breakpoint:
    (gdb) c
    Continuing.
    Hi ya!

    Breakpoint 2, main () at hello.c:11
    11      x=3;

# Notice that the above command went past the
# printf statement at line 9. I/O from the
# printf statement is displayed on screen.

# Inspect variable y, using the short form of the
# inspect command:
    (gdb) ins y
    $1 = -1338755812

# Get some help on the step and next commands:
    (gdb) help s
    Step program until it reaches a different source line.
    Argument N means do this N times (or till program stops
    for another reason).
    (gdb) help n
    Step program, proceeding through subroutine calls.
    Like the "step" command as long as subroutine calls do not
    happen; when they do, the call is treated as one instruction.
    Argument N means do this N times (or till program stops
```

```
    for another reason).

# Go to the next line of execution:
    (gdb) n
    12      y=2;
    (gdb) n
    13      z=3*2;
    (gdb) inspect z
    $2 = 1
    (gdb) n
    15      exit (0);
    (gdb) inspe z
    $3 = 6

# Continue program execution:
    (gdb) continue
    Continuing.

    Program exited normally.

# Quit the debugger session:
    (gdb) quit
    The program is running. Exit anyway? (y or n) y
    (61) con1 /home/allan/src >
```

Sample boot image

```
[virtual=x86,bios +compress] boot = {
    startup-bios -N node428
    PATH=/proc/boot:./ procnto
}

[+script] startup-script = {
# explicitly running in edited mode for the console link
    devc-ser8250 -e -b115200 &
    reopen
    display_msg Welcome to Neutrino on a PC-compatible BIOS system
# tcp/ip with a NE2000 Ethernet adaptor
    io-pkt-v4 -dne2000 -ptcpip if=ndi0:10.0.1.172 &
    waitfor /dev/socket
    pipe &
# pdebug needs devc-pty
    devc-pty &
# starting pdebug twice on separate ports
    [+session] pdebug 8000 &
}

[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[type=link] /lib=/x86/lib
[type=link] /tmp=/dev/shmem         # tmp points to shared memory
[type=link] /dev/console=/dev/ser2  # no local terminal

[perms=+r,+x]          # Boot images made under MS-Windows need
                       # to be reminded of permissions.
devn-ne2000.so
libc.so
fpemu.so
libsocket.so

[data=copy]             # All executables that can be restarted
                        # go below.
devc-ser8250
```

```
io-pkt-v4
pipe
devc-pty
pdebug
esh
ping
ls
```

Debugging using libmudflap

QNX includes support for Mudflap through **libmudflap**. Mudflap provides you with pointer checking capabilities based on compile time instrumentation as it transparently includes protective code to potentially unsafe C/C++ constructs at run time.

For information about the available options for this feature, see the GNU website at:

`http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Optimize-Options.html#index-fmudflap-502`

For more debugging information, you can search the GNU website for the topic "Mudflap Pointer Debugging".

This debugging feature is enabled by passing the option **-fmudflap** to the compiler. For front ends that support it , it instruments all risky pointer and array dereferencing operations, some standard library string and heap functions, and some associated constructs with range and validity tests.

The instrumentation relies on a separate runtime library (**libmudflap**), which is linked into a program if **-fmudflap -lmudflap** is given at link time. Runtime behavior of the instrumented program is controlled by the environment variable >**MUDFLAP_OPTIONS** . You can obtain a list of options by setting **MUDFLAP_OPTIONS** to **-help** and calling a Mudflap compiled program.

For your multithreaded programs:

- To compile, you must use the option **-fmudflapth** instead of **-fmudflap**
- To link, you must use the option **-fmudflapth -lmudflapth**

Additionally, if you want instrumentation to ignore pointer reads, you'll need to use the option **-fmudflapir** in addition to the option **-fmudflap** or **-fmudflapth** (for multithreaded). This option creates less instrumentation, resulting in faster execution.

Regardless of whether you're using **qcc** or **gcc**, for both the compile and link steps you must specify the option **-fmudflap** or **-fmudflapth**.

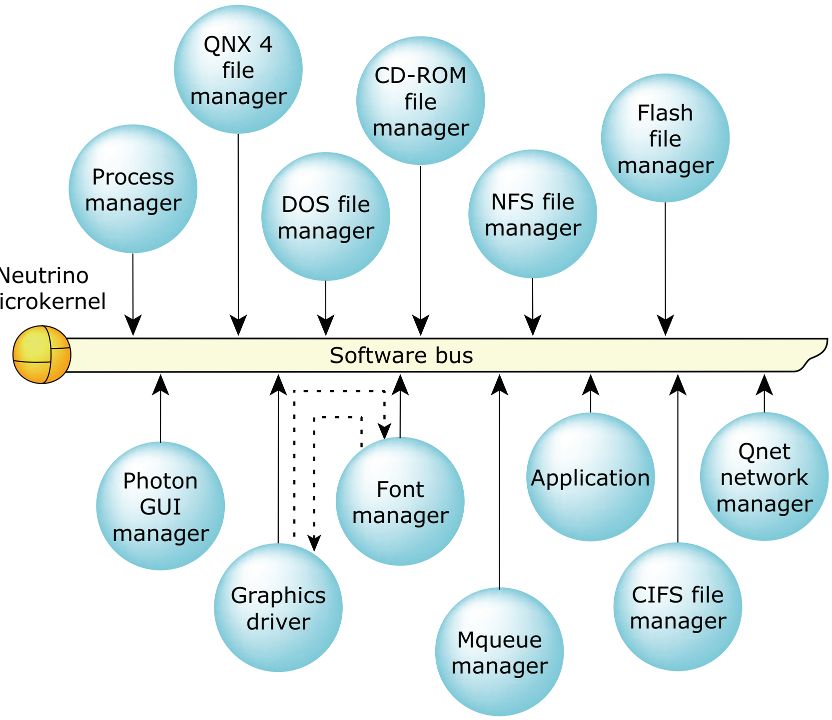
Chapter 2

Programming Overview

In this chapter. . .

Process model

The Neutrino OS architecture consists of the microkernel and some number of cooperating processes. These processes communicate with each other via various forms of interprocess communication (IPC). Message passing is the primary form of IPC in Neutrino.



The Neutrino architecture acts as a kind of "software bus" that lets you dynamically plug in/out OS modules. This picture shows the graphics driver sending a message to the font manager when it wants the bitmap for a font. The font manager responds with the bitmap.

The Photon microGUI windowing system is also made up of a number of cooperating processes: the GUI manager (**`Photon`**), a font manager (**`phfontFA`**), the graphics driver manager (**`io-graphics`**), and others. If the graphics driver needs to draw some text, it sends a message to the font manager asking for bitmaps in the desired font for the text to be drawn in. The font manager responds with the requested bitmaps, and the graphics driver then draws the bitmaps on the screen.

An application as a set of processes

This idea of using a set of cooperating processes isn't limited to the OS "system processes." Your applications should be written in exactly the same way. You might have some driver process that gathers data from some hardware and then needs to pass that data on to other processes, which then act on that data.

Let's use the example of an application that's monitoring the level of water in a reservoir. Should the water level rise too high, then you'll want to alert an operator as well as open some flow-control valve.

In terms of hardware, you'll have some water-level sensor tied to an I/O board in a computer. If the sensor detects some water, it will cause the I/O board to generate an interrupt.

The software consists of a driver process that talks to the I/O board and contains an *interrupt handler* to deal with the board's interrupt. You'll also have a GUI process that will display an alarm window when told to do so by the driver, and finally, another driver process that will open/close the flow-control valve.

Why break this application into multiple processes? Why not have everything done in one process? There are several reasons:

1. Each process lives in its own *protected memory space*. If there's a bug such that a pointer has a value that isn't valid for the process, then when the pointer is next used, the hardware will generate a fault, which the kernel handles (the kernel will set the SIGSEGV signal on the process).

   This approach has two benefits. The first is that a stray pointer won't cause one process to overwrite the memory of another process. The implications are that one process can go bad *while other processes keep running*.

   The second benefit is that the fault will occur precisely when the pointer is used, not when it's overwriting some other process's memory. If a pointer were allowed to overwrite another process's memory, then the problem wouldn't manifest itself until later and would therefore be much harder to debug.

2. It's very easy to add or remove processes from an application as need be. This implies that applications can be made scalable — adding new features is simply a matter of adding processes.

3. Processes can be started and stopped *on the fly*, which comes in handy for dynamic upgrading or simply for stopping an offending process.

4. Processing can be easily distributed across multiple processors in a networked environment.

5. The code for a process is much simpler if it concentrates on doing a single job. For example, a single process that acts as a driver, a GUI front-end, and a data logger would be fairly complex to build and maintain. This complexity would increase the chances of a bug, and any such bug would likely affect all the activities being done by the process.

6. Different programmers can work on different processes without fear of overwriting each other's work.

Processes and threads

Different operating systems often have different meanings for terms such as "process," "thread," "task," "program," and so on.

Some definitions

In the Neutrino OS, we typically use only the terms *process* and *thread*. An "application" typically means a collection of processes; the term "program" is usually equivalent to "process."

A *thread* is a single flow of execution or control. At the lowest level, this equates to the program counter or instruction pointer register advancing through some machine instructions. Each thread has its own current value for this register.

A *process* is a collection of one or more threads that share many things. Threads within a process share at least the following:

- variables that aren't on the stack
- signal handlers (although you typically have one thread that handles signals, and you block them in all the other threads)
- signal ignore mask
- channels
- connections

Threads don't share such things as stack, values for the various registers, SMP thread-affinity mask, and a few other things.

Two threads residing in two different processes don't share very much. About the only thing they do share is the CPU. You can have them share memory between them, but this takes a little setup (see *shm_open()* in the *Library Reference* for an example).

When you run a process, you're automatically running a thread. This thread is called the "main" thread, since the first programmer-provided function that runs in a C program is *main()*. The main thread can then create additional threads if need be.

Only a few things are special about the main thread. One is that if it returns normally, the code it returns to calls *exit()*. Calling *exit()* terminates the process, meaning that all threads in the process are terminated. So when you return normally from the main thread, the process is terminated. When other threads in the process return normally, the code they return to calls *pthread_exit()*, which terminates just that thread.

Another special thing about the main thread is that if it terminates in such a manner that the process is still around (e.g. it calls *pthread_exit()* and there are other threads in the process), then the memory for the main thread's stack is *not* freed up. This is because the command-line arguments are on that stack and other threads may need them. If any other thread terminates, then that thread's stack is freed.

Priorities and scheduling

Although there's a good discussion of priorities and scheduling policies in the *System Architecture* manual (see "Thread scheduling" in the chapter on the microkernel), it will help to go over that topic here in the context of a programmer's guide.

Neutrino provides a priority-driven preemptive architecture. *Priority-driven* means that each thread can be given a priority and will be able to access the CPU based on that priority. If a low-priority thread and a high-priority thread both want to run, then the high-priority thread will be the one that gets to run.

Preemptive means that if a low-priority thread is currently running and then a high-priority thread suddenly wants to run, then the high-priority thread will take over the CPU and run, thereby preempting the low-priority thread.

Priority range

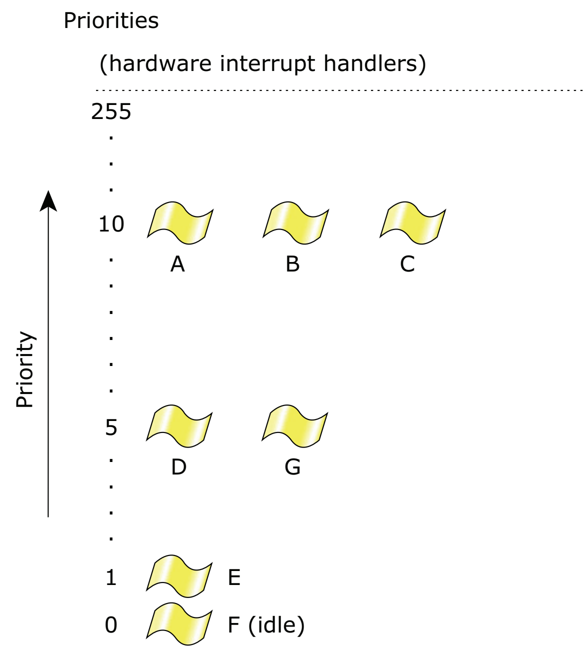
Threads can have a scheduling priority ranging from 1 to 255 (the highest priority), *independent of the scheduling policy*. Non-**`root`** threads can have a priority ranging from 1 to 63 (by default); **`root`** threads (i.e. those with an effective *uid* of 0) are allowed to set priorities above 63.

The special *idle* thread (in the process manager) has priority 0 and is always ready to run. A thread inherits the priority of its parent thread by default.

A thread has both a *real priority* and an *effective priority*, and is scheduled in accordance with its effective priority. The thread itself can change both its real and effective priority together, but the effective priority may change because of priority inheritance or the scheduling policy. Normally, the effective priority is the same as the real priority.

Interrupt handlers are of higher priority than any thread, but they're not scheduled in the same way as threads. If an interrupt occurs, then:

1. Whatever thread was running loses the CPU handling the interrupt (SMP issues).
2. The hardware runs the kernel.
3. The kernel calls the appropriate interrupt handler.



Thread priorities range from 0 (lowest) to 255 (highest). Although interrupt handlers aren't scheduled in the same way as threads, they're considered to be of a higher priority because an interrupt handler will preempt any running thread.

BLOCKED and READY states

To fully understand how scheduling works, you must first understand what it means when we say a thread is BLOCKED and when a thread is in the READY state. You must also understand a particular data structure in the kernel called the *ready queue*.

A thread is BLOCKED if it doesn't want the CPU, which might happen for several reasons, such as:

- The thread is sleeping.
- The thread is waiting for a message from another thread.
- The thread is waiting on a mutex that some other thread owns.

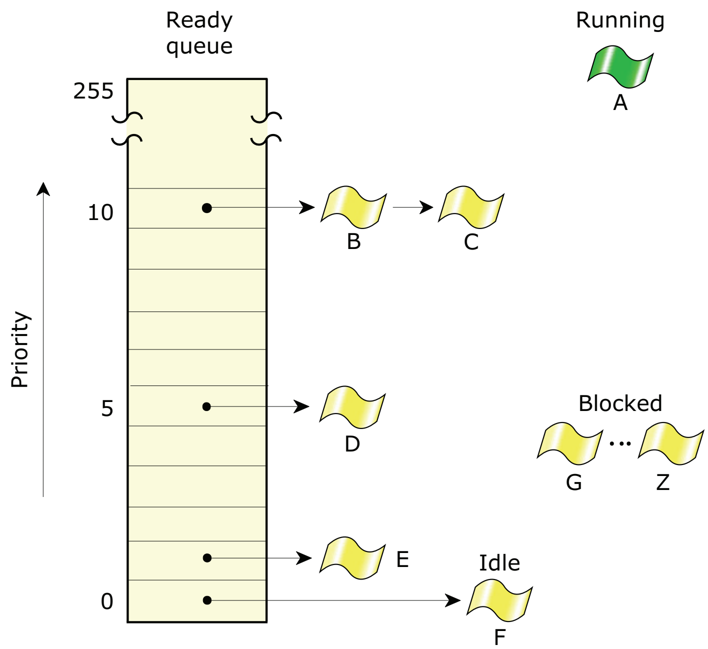
When designing an application, you always try to arrange it so that if any thread is waiting for something, make sure it *isn't spinning in a loop using up the CPU*. In general, try to avoid polling. If you do have to poll, then you should try to sleep for some period between polls, thereby giving lower-priority threads the CPU should they want it.

For each type of blocking there is a blocking state. We'll discuss these states briefly as they come up. Examples of some blocking states are REPLY-blocked, RECEIVE-blocked, MUTEX-blocked, INTERRUPT-blocked, and NANOSLEEP-blocked.

A thread is READY if it wants a CPU but something else currently has it. If a thread currently has a CPU, then it's in the RUNNING state. Simply put, a thread that's either READY or RUNNING isn't blocked.

The ready queue

The ready queue is a simplified version of a kernel data structure consisting of a queue with one entry per priority. Each entry in turn consists of another queue of the threads that are READY at the priority. Any threads that aren't READY aren't in any of the queues — but they will be when they become READY.



The ready queue for five threads (B–F) that are READY. Thread A is currently running. All other threads (G–Z) are BLOCKED. Threads A, B, and C are at the highest priority, so they'll share the processor based on the running thread's scheduling algorithm.

The *active* thread is the one in the RUNNING state. The kernel uses an array (with one entry per processor in the system) to keep track of the running threads.

Every thread is assigned a priority. The scheduler selects the next thread to run by looking at the priority assigned to every thread in the READY state (i.e. capable of using the CPU). The thread with the highest priority that's at the head of its priority's queue is selected to run. In the above diagram, thread A was formerly at the head of priority 10's queue, so thread A was moved to the RUNNING state.

Suspending a running thread

The execution of a running thread is temporarily suspended whenever the microkernel is entered as the result of a kernel call, exception, or hardware interrupt. A scheduling decision is made whenever the execution state of any thread changes — it doesn't matter which processes the threads might reside within. *Threads are scheduled globally across all processes.*

Normally, the execution of the suspended thread will resume, but the scheduler will perform a context switch from one thread to another whenever the running thread:

- is blocked
- is preempted
- yields

When the thread is blocked

The running thread will block when it must wait for some event to occur (response to an IPC request, wait on a mutex, etc.). The blocked thread is removed from the running array, and the highest-priority ready thread that's at the head of its priority's queue is then allowed to run. When the blocked thread is subsequently unblocked, it's placed on the end of the ready queue for its priority level.

When the thread is preempted

The running thread will be preempted when a higher-priority thread is placed on the ready queue (it becomes READY as the result of its block condition being resolved). The preempted thread is moved to the start of the ready queue for that priority, and the higher-priority thread runs. When it's time for a thread at that priority level to run again, that thread resumes execution — a preempted thread will not lose its place in the queue for its priority level.

When the thread yields

The running thread voluntarily yields the processor (via *sched_yield()*) and is placed on the end of the ready queue for that priority. The highest-priority thread then runs (which may still be the thread that just yielded).

Scheduling algorithms

To meet the needs of various applications, Neutrino provides these scheduling algorithms:

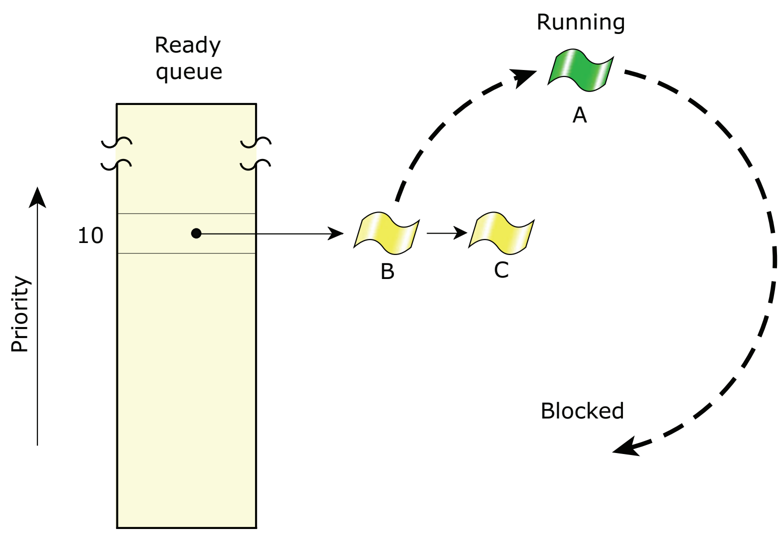
- FIFO scheduling — SCHED_FIFO
- Round-robin scheduling — SCHED_RR
- Sporadic scheduling — SCHED_SPORADIC

Another scheduling algorithm (called “other” — SCHED_OTHER) behaves in the same way as round-robin. We don’t recommend using the “other” scheduling algorithm, because its behavior may change in the future.

Each thread in the system may run using any method. Scheduling methods are effective on a per-thread basis, not on a global basis for all threads and processes on a node.

Remember that these scheduling algorithms apply only when two or more threads that share the same priority are READY (i.e. the threads are directly competing with each other). If a higher-priority thread becomes READY, it immediately preempts all lower-priority threads.

In the following diagram, three threads of equal priority are READY. If Thread A blocks, Thread B will run.



Thread A blocks; Thread B runs.

A thread can call *pthread_attr_setschedparam()* or *pthread_attr_setschedppolicy()* to set the scheduling parameters and policy to use for any threads that it creates.

Although a thread inherits its scheduling algorithm from its parent thread, the thread can call *pthread_setschedparam()* to request to change the algorithm and priority applied by the kernel. A thread can get information about its current algorithm and policy by calling *pthread_getschedparam()*. Both these functions take a thread ID as their first argument; you can call *pthread_self()* to get the calling thread’s ID. For example:

```
struct sched_param param;
int policy, retcode;

/* Get the scheduling parameters. */
```

```
retcode = pthread_getschedparam( pthread_self(), &policy, &param);
if (retcode != EOK) {
    printf ("pthread_getschedparam: %s.\n", strerror (retcode));
    return EXIT_FAILURE;
}

printf ("The assigned priority is %d, and the current priority is %d.\n",
        param.sched_priority, param.sched_curpriority);

/* Increase the priority. */

param.sched_priority++;

retcode = pthread_setschedparam( pthread_self(), policy, &param);
if (retcode != EOK) {
    printf ("pthread_setschedparam: %s.\n", strerror (retcode));
    return EXIT_FAILURE;
}
```

When you get the scheduling parameters, the *sched_priority* member of the **sched_param** structure is set to the assigned priority, and the *sched_curpriority* member is set to the priority that the thread is currently running at (which could be different because of priority inheritance).

Our libraries provide a number of ways to get and set scheduling parameters:

pthread_getschedparam(), *pthread_setschedparam()*
: These are your best choice for portability.

SchedGet(), *SchedSet()*
: You can use these to get and set the scheduling priority and policy, but they aren't portable because they're kernel calls.

sched_getparam(), *sched_setparam()*, *sched_getscheduler()*, and *sched_setscheduler()*,
: These functions are intended for use in single-threaded processes.

Our implementations of these functions don't conform completely to POSIX. In multi-threaded applications, they get or set the parameters for thread 1 in the process *pid*, or for the *calling thread* if *pid* is 0. If you depend on this behavior, your code won't be portable. POSIX 1003.1 says these functions should return -1 and set *errno* to EPERM in a multi-threaded application.

---

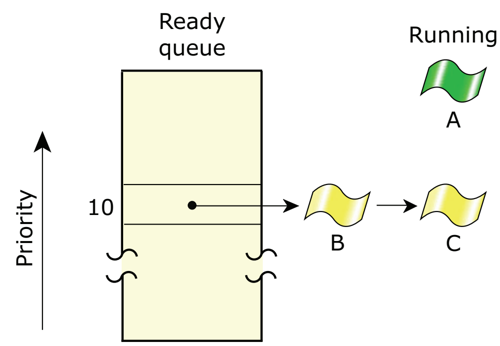
getprio(), *setprio()*
: QNX Neutrino supports these functions only for compatibility with QNX 4 programs; don't use them in new programs.

getpriority(), *setpriority()*
: Deprecated; don't use these functions.

FIFO scheduling

In FIFO (SCHED_FIFO) scheduling, a thread selected to run continues executing until it:

- voluntarily relinquishes control (e.g. it blocks)
- is preempted by a higher-priority thread

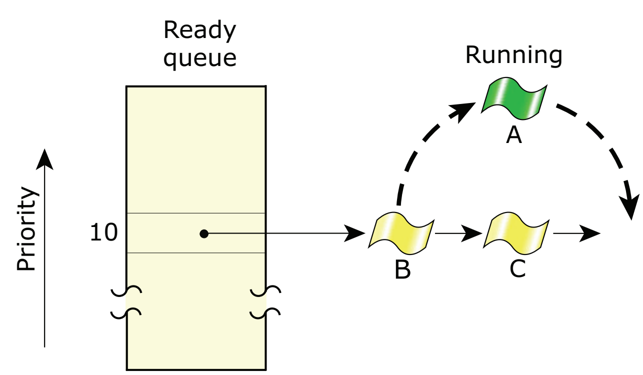


FIFO scheduling. Thread A runs until it blocks.

Round-robin scheduling

In round-robin (SCHED_RR) scheduling, a thread selected to run continues executing until it:

- voluntarily relinquishes control
- is preempted by a higher-priority thread
- consumes its timeslice



Round-robin scheduling. Thread A ran until it consumed its timeslice; the next READY thread (Thread B) now runs.

A *timeslice* is the unit of time assigned to every process. Once it consumes its timeslice, a thread is put at the end of its queue in the ready queue and the next READY thread at the same priority level is given control.

A timeslice is calculated as:

$4 \times ticksize$

If your processor speed is greater than 40 MHz, then the ticksize defaults to 1 millisecond; otherwise, it defaults to 10 milliseconds. So, the default timeslice is either 4 milliseconds (the default for most CPUs) or 40 milliseconds (the default for slower hardware).

Apart from time-slicing, the round-robin scheduling method is identical to FIFO scheduling.

Sporadic scheduling

The sporadic (SCHED_SPORADIC) scheduling algorithm is generally used to provide a capped limit on the execution time of a thread *within a given period of time*. This behavior is essential when Rate Monotonic Analysis (RMA) is being performed on a system that services both periodic and aperiodic events. Essentially, this algorithm allows a thread to service aperiodic events without jeopardizing the hard deadlines of other threads or processes in the system.

Under sporadic scheduling, a thread's priority can oscillate dynamically between a *foreground* or normal priority and a *background* or low priority. For more information, see "Sporadic scheduling" in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

Why threads?

Now that we know more about priorities, we can talk about why you might want to use threads. We saw many good reasons for breaking things up into separate processes, but what's the purpose of a *multithreaded* process?

Let's take the example of a driver. A driver typically has two obligations: one is to talk to the hardware and the other is to talk to other processes. Generally, talking to the hardware is more time-critical than talking to other processes. When an interrupt comes in from the hardware, it needs to be serviced in a relatively small window of time — the driver shouldn't be busy at that moment talking to another process.

One way of fixing this problem is to choose a way of talking to other processes where this situation simply won't arise (e.g. don't send messages to another process such that you have to wait for acknowledgment, don't do any time-consuming processing on behalf of other processes, etc.).

Another way is to use two threads: a higher-priority thread that deals with the hardware and a lower-priority thread that talks to other processes. The lower-priority thread can be talking away to other processes without affecting the time-critical job at all, because when the interrupt occurs, the *higher-priority thread will preempt the lower-priority thread* and then handle the interrupt.

Although this approach does add the complication of controlling access to any common data structures between the two threads, Neutrino provides synchronization tools such as *mutexes* (mutual exclusion locks), which can ensure exclusive access to any data shared between threads.

Summary

The modular architecture is apparent throughout the entire system: the Neutrino OS itself consists of a set of cooperating processes, as does an application. And each individual process can comprise several cooperating threads. What “keeps everything together” is the priority-based preemptive scheduling in Neutrino, which ensures that time-critical tasks are dealt with by the right thread or process at the right time.

Chapter 3

Processes

In this chapter. . .

As we stated in the Overview chapter, the Neutrino OS architecture consists of a small microkernel and some number of cooperating *processes*. We also pointed out that your applications should be written the same way — as a set of cooperating processes.

In this chapter, we'll see how to start processes (also known as *creating* processes) from code, how to terminate them, and how to detect their termination when it happens.

For another perspective, see the Processes and Threads and Message Passing chapters of *Getting Started with QNX Neutrino*.

Starting processes — two methods

In embedded applications, there are two typical approaches to starting your processes at boot time. One approach is to run a *shell script* that contains the command lines for running the processes. There are some useful utilities such as **`on`** and **`nice`** for controlling how those processes are started.

The other approach is to have a *starter process* run at boot time. This starter process then starts up all your other processes. This approach has the advantage of giving you more control over how processes are started, whereas the script approach is easier for you (or anyone) to modify quickly.

Process creation

The process manager component of **`procnto`** is responsible for process creation. If a process wants to create another process, it makes a call to one of the process-creation functions, which then effectively sends a message to the process manager.

Here are the process-creation functions:

- *exec*()* family of functions: *execl()*, *execle()*, *execlp()*, *execlpe()*, *execv()*, *execve()*, *execvp()*, *execvpe()*
- *fork()*
- *forkpty()*
- *popen()*
- *spawn()*
- *spawn*()* family of functions: *spawn()*, *spawnl()*, *spawnle()*, *spawnlp()*, *spawnlpe()*, *spawnp()*, *spawnv()*, *spawnve()*, *spawnvp()*, *spawnvpe()*
- *system()*
- *vfork()*

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

- You specify `P_OVERLAY` when calling one of the *spawn** functions.

- You call one of the *exec** routines.

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

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

There are several versions of *spawn*()* and *exec*()*. The `*` is one to three letters, where:

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

For details on each of these functions, see their entries in the *Library Reference*. Here we'll mention some of the things common to many of them.

Concurrency

Three possibilities can happen to the creator during process creation:

1. The child process is created and runs concurrently with the parent. In this case, as soon as process creation is successful, the process manager replies to the parent, and the child is made READY. If it's the parent's turn to run, then the first thing it does is return from the process-creation function. This may not be the case if the child process was created at a higher priority than the parent (in which case the child will run before the parent gets to run again).

   This is how *fork()*, *forkpty()*, *popen()*, and *spawn()* work. This is also how the *spawn*()* family of functions work when the mode is passed as P_NOWAIT or P_NOWAITO.

2. The child replaces the parent. In fact, they're not really parent and child, because the image of the given process simply replaces that of the caller. Many things will change, but those things that uniquely identify a process (such as the process ID) will remain the same. This is typically referred to as "execing," since usually the *exec*()* functions are used.

   Many things will remain the same (including the process ID, parent process ID, and file descriptors) with the exception of file descriptors that had the FD_CLOEXEC flag set using *fcntl()*. See the *exec*()* functions for more on what will and will not be the same across the exec.

   The **`login`** command serves as a good example of execing. Once the login is successful, the **`login`** command execs into a shell.

   Functions you can use for this type of process creation are the *exec*()* and *spawn*()* families of functions, with mode passed as P_OVERLAY.

3. The parent waits until the child terminates. This can be done by passing the mode as P_WAIT for the *spawn*()* family of functions.

   Note that what is going on underneath the covers in this case is that *spawn()* is called as in the first possibility above. Then, after it returns, *waitpid()* is called in order to wait for the child to terminate. This means that you can use any of the functions mentioned in our first possibility above to achieve the same thing if you follow them by a call to one of the *wait*()* functions (e.g. *wait()* or *waitpid()*).

Using *fork()* and *forkpty()*

As of this writing, you can't use *fork()* and *forkpty()* in a process that has threads. The *fork()* and *forkpty()* functions will simply return -1 and *errno* will be set to ENOSYS.

Many programmers coming from the Unix world are familiar with the technique of using a call to *fork()* followed by a call to one of the *exec*()* functions in order to create a process that's different from the caller. In Neutrino, you can usually achieve the same thing in a single call to one of the *spawn*()* functions.

Inheriting file descriptors

The documentation in the QNX Neutrino *Library Reference* for each function describes in detail what the child inherits from the parent. One thing that we should talk about here, however, is file-descriptor inheritance.

With many of the process-creation functions, the child inherits the file descriptors of the parent. For example, if the parent had file descriptor 5 in use for a particular file when the parent creates the child, the child will also have file descriptor 5 in use for that same file. The child's file descriptor will have been duplicated from the parent's. This means that at the filesystem manager level, the parent and child have the same open control block (OCB) for the file, so if the child seeks to some position in the file, then that changes the parent's seek position as well. It also means that the child can do a **write(5, buf, nbytes)** without having previously called *open()*.

If you don't want the child to inherit a particular file descriptor, then you can use *fcntl()* to prevent it. Note that this won't prevent inheritance of a file descriptor during a *fork()*. The call to *fcntl()* would be:

```
fcntl(fd, F_SETFD, FD_CLOEXEC);
```

If you want the parent to set up exactly which files will be open for the child, then you can use the *fd_count* and *fd_map* parameters with *spawn()*. Note that in this case, only the file descriptors you specify will be inherited. This is especially useful for redirecting the child's standard input (file descriptor 0), standard output (file descriptor 1), and standard error (file descriptor 2) to places where the parent wants them to go.

Alternatively this file descriptor inheritance can also be done through use of *fork()*, one or more calls to *dup()*, *dup2()*, and *close()*, and then *exec*()*. The call to *fork()* creates a child that inherits all the of the parent's file descriptors. *dup()*, *dup2()* and *close()* are then used by the child to rearrange its file descriptors. Lastly, *exec*()* is called to replace the child with the process to be created. Though more complicated, this method of setting up file descriptors is portable whereas the *spawn()* method is not.

Process termination

A process can terminate in one of two basic ways:

- normally (e.g. the process terminates itself)
- abnormally (e.g. the process terminates as the result of a signal's being set)

Normal process termination

A process can terminate itself by having any thread in the process call *exit()*. Returning from the main thread (i.e. *main()*) will also terminate the process, because the code that's returned to calls *exit()*. This isn't true of threads other than the main thread. Returning normally from one of them causes *pthread_exit()* to be called, which terminates only that thread. Of course, if that thread is the last one in the process, then the process is terminated.

The value passed to *exit()* or returned from *main()* is called the *exit status*.

Abnormal process termination

A process can be terminated abnormally for a number of reasons. Ultimately, all of these reasons will result in a *signal's being set on the process*. A signal is something that can interrupt the flow of your threads at any time. The default action for most signals is to terminate the process.

Note that what causes a particular signal to be generated is sometimes processor-dependent.

Here are some of the reasons that a process might be terminated abnormally:

- If any thread in the process tries to use a pointer that doesn't contain a valid virtual address for the process, then the hardware will generate a fault and the kernel will handle the fault by setting the SIGSEGV signal on the process. By default, this will terminate the process.
- A floating-point exception will cause the kernel to set the SIGFPE signal on the process. The default is to terminate the process.
- If you create a shared memory object and then map in more than the size of the object, when you try to write past the size of the object you'll be hit with SIGBUS. In this case, the virtual address used is valid (since the mapping succeeded), but the memory cannot be accessed.

To get the kernel to display some diagnostics whenever a process terminates abnormally, configure **procnto** with multiple **-v** options. If the process has fd 2 open, then the diagnostics are displayed using (*stderr*); otherwise; you can specify where the diagnostics get displayed by using the **-D** option to your startup. For example, the **-D** as used in this buildfile excerpt will cause the output to go to a serial port:

```
[virtual=x86,bios +compress] .bootstrap = {
    startup-bios -D 8250..115200
    procnto -vvvv
}
```

You can also have the current state of a terminated process written to a file so that you can later bring up the debugger and examine just what happened. This type of examination is called *postmortem* debugging. This happens only if the process is terminated due to one of these signals:

Signal	Description
SIGABRT	Program-called abort function
SIGBUS	Parity error
SIGEMT	EMT instruction
SIGFPE	Floating-point error or division by zero
SIGILL	Illegal instruction executed
SIGQUIT	Quit
SIGSEGV	Segmentation violation
SIGSYS	Bad argument to a system call
SIGTRAP	Trace trap (not reset when caught)
SIGXCPU	Exceeded the CPU limit
SIGXFSZ	Exceeded the file size limit

The process that dumps the state to a file when the process terminates is called **dumper**, which must be running when the abnormal termination occurs. This is extremely useful, because embedded systems may run unassisted for days or even years before a crash occurs, making it impossible to reproduce the actual circumstances leading up to the crash.

Effect of parent termination

In some operating systems, if a parent process dies, then all of its child processes die too. This isn't the case in Neutrino.

Detecting process termination

In an embedded application, it's often important to detect if any process terminates prematurely and, if so, to handle it. Handling it may involve something as simple as restarting the process or as complex as:

1. Notifying other processes that they should put their systems into a safe state.
2. Resetting the hardware.

This is complicated by the fact that some Neutrino processes call *procmgr_daemon()*. Processes that call this function are referred to as *daemons*. The *procmgr_daemon()* function:

- detaches the caller from the controlling terminal
- puts it in session 1
- optionally, closes all file descriptors except *stdin*, *stdout*, and *stderr*
- optionally, redirects *stdin*, *stdout*, *stderr* to **`/dev/null`**

As a result of the above, their termination is hard to detect.

Another scenario is where a server process wants to know if any of its clients disappear so that it can clean up any resources it had set aside on their behalf.

Let's look at various ways of detecting process termination.

Using the High Availability Framework

The High Availability Framework provides components not only for detecting when processes terminate, but also for recovering from that termination.

The main component is a process called the High Availability Manager (HAM) that acts as a "smart watchdog". Your processes talk to the HAM using the HAM API. With this API you basically set up conditions that the HAM should watch for and take actions when these conditions occur. So the HAM can be told to detect when a process terminates and to automatically restart the process. It will even detect the termination of daemon processes.

In fact, the High Availability Manager can restart a number of processes, wait between restarts for a process to be ready, and notify the process that this is happening.

The HAM also does heartbeating. Processes can periodically notify the HAM that they are still functioning correctly. If a process specified amount of time goes by between these notifications then the HAM can take some action.

The above are just a sample of what is possible with the High Availability Framework. For more information, see the High Availability Framework *Developer's Guide*

Detecting termination from a starter process

If you've created a set of processes using a starter process as discussed at the beginning of this section, then all those processes are children of the starter process, with the exception of those that have called *procmgr_daemon()*. If all you want to do is detect that one of those children has terminated, then a loop that blocks on *wait()* or *sigwaitinfo()* will suffice. Note that when a child process calls *procmgr_daemon()*, both *wait()* and *sigwaitinfo()* behave as if the child process died, although the child is still running.

The *wait()* function will block, waiting until any of the caller's child processes terminate. There's also *waitpid()*, which lets you wait for a specific child process, *wait3()*, and *wait4()*. Lastly, there is *waitid()*, which is the lower level of all the *wait*()* functions and returns the most information.

The *wait*()* functions won't always help, however. If a child process was created using one of the *spawn*()* family of functions with the mode passed as P_NOWAITO, then the *wait*()* functions won't be notified of its termination!

What if the child process terminates, but the parent hasn't yet called *wait*()*? This would be the case if one child had already terminated, so *wait*()* returned, but then before the parent got back to the *wait*()*, *a second child terminates*. In that case, some information would have to be stored away about the second child for when the parent does get around to its *wait*()*.

This is in fact the case. The second child's memory will have been freed up, its files will have been closed, and in general the child's resources will have been cleaned up with the exception of a few bytes of memory in the process manager that contain the child's exit status or other reason that it had terminated and its process ID. When the second child is in this state, it's referred to as a *zombie*. The child will remain a zombie until the parent either terminates or finds out about the child's termination (e.g. the parent calls *wait*()*).

What this means is that if a child has terminated and the parent is still alive but doesn't yet know about the terminated child (e.g. hasn't called *wait*()*), then the zombie will be hanging around. If the parent will never care, then you may as well not have the child become a zombie. To prevent the child from becoming a zombie when it terminates, create the child process using one of the *spawn*()* family of functions and pass P_NOWAITO for the *mode*.

Sample parent process using *wait()*

The following sample illustrates the use of *wait()* for waiting for child processes to terminate.

```
/*
 * waitchild.c
 *
 * This is an example of a parent process that creates some child
 * processes and then waits for them to terminate. The waiting is
 * done using wait(). When a child process terminates, the
 * wait() function returns.
*/

#include <spawn.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

main(int argc, char **argv)
{
    char                *args[] = { "child", NULL };
    int                 i, status;
    pid_t               pid;
    struct inheritance  inherit;

    // create 3 child processes
    for (i = 0; i < 3; i++) {
```

```
        inherit.flags = 0;
        if ((pid = spawn("child", 0, NULL, &inherit, args, environ)) == -1)
            perror("spawn() failed");
        else
            printf("spawned child, pid = %d\n", pid);
    }

    while (1) {
        if ((pid = wait(&status)) == -1) {
            perror("wait() failed (no more child processes?)");
            exit(EXIT_FAILURE);
        }
        printf("a child terminated, pid = %d\n", pid);

        if (WIFEXITED(status)) {
            printf("child terminated normally, exit status = %d\n",
                WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("child terminated abnormally by signal = %X\n",
                WTERMSIG(status));
        } // else see documentation for wait() for more macros
    }
}
```

The following is a simple child process to try out with the above parent.

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

main(int argc, char **argv)
{
    printf("pausing, terminate me somehow\n");
    pause();
}
```

The *sigwaitinfo()* function will block, waiting until any signals that the caller tells it to wait for are set on the caller. If a child process terminates, then the SIGCHLD signal is set on the parent. So all the parent has to do is request that *sigwaitinfo()* return when SIGCHLD arrives.

Sample parent process using *sigwaitinfo()*

The following sample illustrates the use of *sigwaitinfo()* for waiting for child processes to terminate.

```
/*
 * sigwaitchild.c
 *
 * This is an example of a parent process that creates some child
 * processes and then waits for them to terminate.  The waiting is
 * done using sigwaitinfo().  When a child process terminates, the
 * SIGCHLD signal is set on the parent.  sigwaitinfo() will return
 * when the signal arrives.
*/

#include <errno.h>
#include <spawn.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/neutrino.h>

void
signal_handler(int signo)
{
    // do nothing
}
```

```
main(int argc, char **argv)
{
    char                *args[] = { "child", NULL };
    int                 i;
    pid_t               pid;
    sigset_t            mask;
    siginfo_t           info;
    struct inheritance  inherit;
    struct sigaction    action;

    // mask out the SIGCHLD signal so that it will not interrupt us,
    // (side note: the child inherits the parents mask)
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    // by default, SIGCHLD is set to be ignored so unless we happen
    // to be blocked on sigwaitinfo() at the time that SIGCHLD
    // is set on us we will not get it.  To fix this, we simply
    // register a signal handler.  Since we've masked the signal
    // above, it will not affect us.  At the same time we will make
    // it a queued signal so that if more than one are set on us,
    // sigwaitinfo() will get them all.
    action.sa_handler = signal_handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = SA_SIGINFO; // make it a queued signal
    sigaction(SIGCHLD, &action, NULL);

    // create 3 child processes
    for (i = 0; i < 3; i++) {
        inherit.flags = 0;
        if ((pid = spawn("child", 0, NULL, &inherit, args, environ)) == -1)
            perror("spawn() failed");
        else
            printf("spawned child, pid = %d\n", pid);
    }

    while (1) {
        if (sigwaitinfo(&mask, &info) == -1) {
            perror("sigwaitinfo() failed");
            continue;
        }
        switch (info.si_signo) {
        case SIGCHLD:
            // info.si_pid is pid of terminated process, it is not POSIX
            printf("a child terminated, pid = %d\n", info.si_pid);
            break;
        default:
            // should not get here since we only asked for SIGCHLD
        }
    }
}
```

Detecting dumped processes

As mentioned above, you can run **dumper** so that when a process dies, **dumper** writes the state of the process to a file.

You can also write your own dumper-type process to run instead of, or as well as, **dumper**. This way the terminating process doesn't have to be a child of yours.

To do this, write a resource manager that registers the name, **/proc/dumper** with type _FTYPE_DUMPER. When a process dies due to one of the appropriate signals, the process manager will open **/proc/dumper** and write the pid of the process that died — then it'll wait until you reply to the write with success and then it'll finish terminating the process.

It's possible that more than one process will have **/proc/dumper** registered at the same time, however, the process manager notifies only the process that's at the beginning of its list for that name. Undoubtedly, you want both your resource manager and **dumper** to handle this termination. To do this, request the process manager to put you, instead of **dumper**, at the beginning of the **/proc/dumper** list by passing _RESMGR_FLAG_BEFORE in the *flags* argument to *resmgr_attach()*. You must also open **/proc/dumper** so that you can communicate with **dumper** if it's running. Whenever your io_write handler is called, write the pid to **dumper** and do your own handling. Of course this works only when **dumper** is run before your resource manager; otherwise, your open of **/proc/dumper** won't work.

The following is a sample process that demonstrates the above:

```
/*
 *  dumphandler.c
 *
 *  This demonstrates how you get notified whenever a process
 *  dies due to any of the following signals:
 *
 *  SIGABRT
 *  SIGBUS
 *  SIGEMT
 *  SIGFPE
 *  SIGILL
 *  SIGQUIT
 *  SIGSEGV
 *  SIGSYS
 *  SIGTRAP
 *  SIGXCPU
 *  SIGXFSZ
 *
 *  To do so, register the path, /proc/dumper with type
 *  _FTYPE_DUMPER. When a process dies due to one of the above
 *  signals, the process manager will open /proc/dumper, and
 *  write the pid of the process that died - it will wait until
 *  you reply to the write with success, and then it will finish
 *  terminating the process.
 *
 *  Note that while it is possible for more than one process to
 *  have /proc/dumper registered at the same time, the process
 *  manager will notify only the one that is at the beginning of
 *  its list for that name.
 *
 *  But we want both us and dumper to handle this termination.
 *  To do this, we make sure that we get notified instead of
 *  dumper by asking the process manager to put us at the
 *  beginning of its list for /proc/dumper (done by passing
 *  _RESMGR_FLAG_BEFORE to  resmgr_attach()).  We also open
 *  /proc/dumper so that we can communicate with dumper if it is
 *  running.  Whenever our io_write handler is called, we write
 *  the pid to dumper and do our own handling.  Of course, this
 *  works only if dumper is run before we are, or else our open
 *  will not work.
 *
*/

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

```
#include <string.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>
#include <sys/procfs.h>
#include <sys/stat.h>

int io_write (resmgr_context_t *ctp, io_write_t  *msg,
              RESMGR_OCB_T *ocb);

static int  dumper_fd;

resmgr_connect_funcs_t  connect_funcs;
resmgr_io_funcs_t       io_funcs;
dispatch_t              *dpp;
resmgr_attr_t           rattr;
dispatch_context_t      *ctp;
iofunc_attr_t           ioattr;

char    *progname = "dumphandler";

main(int argc, char **argv)
{
    /* find dumper so that we can pass any pids on to it */
    dumper_fd = open("/proc/dumper", O_WRONLY);

    dpp = dispatch_create();

    memset(&rattr, 0, sizeof(rattr));
    rattr.msg_max_size = 2048;

    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                     _RESMGR_IO_NFUNCS, &io_funcs);
    io_funcs.write = io_write;

    iofunc_attr_init(&ioattr, S_IFNAM | 0600, NULL, NULL);

    resmgr_attach(dpp, &rattr, "/proc/dumper", _FTYPE_DUMPER,
                  _RESMGR_FLAG_BEFORE, &connect_funcs,
                  &io_funcs, &ioattr);

    ctp = dispatch_context_alloc(dpp);

    while (1) {
        if ((ctp = dispatch_block(ctp)) == NULL) {
            fprintf(stderr, "%s:  dispatch_block failed: %s\n",
                             progname, strerror(errno));
            exit(1);
        }
        dispatch_handler(ctp);
    }
}

struct dinfo_s {
    procfs_debuginfo    info;
    char                pathbuffer[PATH_MAX]; /* 1st byte is
                                                 info.path[0] */
};

int
display_process_info(pid_t pid)
{
```

```
    char            buf[PATH_MAX + 1];
    int             fd, status;
    struct dinfo_s  dinfo;
    procfs_greg     reg;

    printf("%s: process %d died\n", progname, pid);

    sprintf(buf, "/proc/%d/as", pid);

    if ((fd = open(buf, O_RDONLY|O_NONBLOCK)) == -1)
        return errno;

    status = devctl(fd, DCMD_PROC_MAPDEBUG_BASE, &dinfo,
                    sizeof(dinfo), NULL);
    if (status != EOK) {
        close(fd);
        return status;
    }

    printf("%s: name is %s\n", progname, dinfo.info.path);

    /*
     * For getting other type of information, see sys/procfs.h,
     * sys/debug.h, and sys/dcmd_proc.h
     */

    close(fd);
    return EOK;
}

int
io_write(resmgr_context_t *ctp, io_write_t *msg,
         RESMGR_OCB_T *ocb)
{
    char    *pstr;
    int     status;

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL))
        != EOK)
        return status;

    if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE)
        return ENOSYS;

    if (ctp->msg_max_size < msg->i.nbytes + 1)
        return ENOSPC; /* not all the message could fit in the
                          message buffer */

    pstr = (char *) (&msg->i) + sizeof(msg->i);
    pstr[msg->i.nbytes] = '\0';

    if (dumper_fd != -1) {
        /* pass it on to dumper so it can handle it too */
        if (write(dumper_fd, pstr, strlen(pstr)) == -1) {
            close(dumper_fd);
            dumper_fd = -1; /* something wrong, no sense in
                               doing it again later */
        }
    }

    if ((status = display_process_info(atoi(pstr))) == -1)
        return status;
```

```
    _IO_SET_WRITE_NBYTES(ctp, msg->i.nbytes);

    return EOK;
}
```

For more information about getting process information (including using the DCMD_PROC_MAPDEBUG_BASE *devctl()* command), see "Controlling processes via the **/proc** filesystem," later in this chapter.

Detecting the termination of daemons

What would happen if you've created some processes that subsequently made themselves daemons (i.e. called *procmgr_daemon()*)? As we mentioned above, the *wait*()* functions and *sigwaitinfo()* won't help.

For these you can give the kernel an event, such as one containing a pulse, and have the kernel deliver that pulse to you whenever a daemon terminates. This request for notification is done by calling *procmgr_event_notify()* with PROCMGR_EVENT_DAEMON_DEATH in *flags*.

See the documentation for *procmgr_event_notify()* for an example that uses this function.

Detecting client termination

The last scenario is where a server process wants to be notified of any clients that terminate so that it can clean up any resources that it had set aside for them.

This is very easy to do if the server process is written as a *resource manager*, because the resource manager's *io_close_dup()* and *io_close_ocb()* handlers, as well as the *ocb_free()* function, will be called if a client is terminated for any reason. For more information, see *Writing a Resource Manager*.

Controlling processes via the /proc filesystem

Implemented by the Process Manager component of **procnto**, the **/proc** virtual filesystem lets you access and control every process and thread running within the system.

The **/proc** filesystem manifests each process currently running on the system as a directory whose name is the numerical process ID (decimal) of the process. Inside this directory, you'll find a file called **as** ("address space") that contains the process's entire memory space. Threads are accessible through the **as** file created for the process; you can select a thread via *devctl()* calls. You can use the following standard functions to access the **/proc** filesystem:

Function	Purpose
open()	Establish a file descriptor to a process
read()	Read data from the process's address space
write()	Write data to the process's address space
stat()	Return **struct stat** information
lseek()	Establish a position within the process's address space for further operations
devctl()	Manipulate a process or thread
close()	Release a file descriptor

Ancillary functions (such as *readdir()*, *opendir()*, and so on) are supported on the directory **/proc** itself — this aids in implementing commands such as **ls**.

Establishing a connection

To be able to access a process or thread, you must first use the *open()* call to get a valid file descriptor. You can them use this file descriptor with the function calls listed below to access the process or thread.

Open the file (**/proc/***pid***/as**), not the **/proc/***pid* directory.

In order to read or write data from or to the process, you must have opened the file descriptor in the appropriate mode. You must also have appropriate privileges to open the particular process. By default:

- Any process can read any other process's address space.
- To write to a process's address space, your user ID and group ID must match that of the process or you must be **root**. Only one process can have a **/proc/***pid***/as** file open for writing at a time.

CAUTION:

The default permissions on these files can be a security problem. When you start **procnto**, you can use the **-u** option to specify the **umask** to use for entries in **/proc/***pid*. The downside of tightening up the **umask** is that some applications (e.g. **pidin arg** and the shelf's network monitor) assume they can open these files.

When you're done accessing the process or thread, you should *close()* the file descriptor. Depending on what you were doing, certain actions can occur on the process or thread when you perform the *close()*. These actions are documented below.

Reading and writing the process's address space

The easiest operation to perform is to access the process's address space. (Since threads exist in the context of a process and have access to everything within a process, there's no need to consider threads in this discussion.)

You can use the *read()*, *write()*, and *lseek()* functions to access the process's address space. The *read()* function transfers bytes from the current position within the process to the program issuing the *read()*, and *write()* transfers bytes from the program to the process.

Determining the offset

The position at which transfer occurs depends on the current offset as set on the file descriptor. In virtual-address systems such as QNX Neutrino, the current offset is taken to be the virtual address *from the process's perspective*.

For example, to read 4096 bytes at offset `0x00021000` from process ID number 2259, the following code snippet could be used:

```
int     fd;
char    buf [4096];

fd = open ("/proc/2259/as", O_RDONLY);
lseek (fd, 0x00021000, SEEK_SET);
read (fd, buf, 4096);
```

Of course, you should check the return values in your real code!

Determining accessibility

If a virtual address process has different chunks of memory mapped into its address space, performing a read or write on a given address may or may not work (or it may not affect the expected number of bytes). This is because the *read()* and *write()* functions affect only *contiguous* memory regions. If you try to read a page of memory that isn't mapped by the process, the read will fail; this is expected.

Manipulating a process or thread

Once you have a file descriptor to a particular process, you can do a number of things to that process and its associated thread(s):

- select a particular thread for further operations
- start and stop a particular thread
- set breakpoints
- examine process and thread attributes (e.g. CPU time)

All of these functions are performed using the *devctl()* call as described in the sections that follow. To be able to use these *devctl()* calls, you'll need at least the following:

```
#include <devctl.h>
#include <sys/procfs.h>
```

Selecting a thread for further operations

When you first perform the *open()* to a particular process, by default you're connected to the first thread (the thread that executed the *main()* function).

If you wish to switch a different thread, use the DCMD_PROC_CURTHREAD *devctl()* command, as described later in this chapter.

To find out how many threads are available in the given process, see the *devctl()* command DCMD_PROC_INFO, below.

Starting/stopping processes and threads

The following *devctl()* commands start and stop processes and threads. You must have opened the file descriptor for writing.

- DCMD_PROC_STOP
- DCMD_PROC_RUN
- DCMD_PROC_FREEZETHREAD
- DCMD_PROC_THAWTHREAD

Setting breakpoints

The following *devctl()* commands set breakpoints. You must have opened the file descriptor for writing.

- DCMD_PROC_BREAK
- DCMD_PROC_WAITSTOP
- DCMD_PROC_GET_BREAKLIST

Examining process and thread attributes

You can use the following *devctl()* commands to examine process and thread attributes:

- DCMD_PROC_SYSINFO
- DCMD_PROC_INFO
- DCMD_PROC_MAPINFO
- DCMD_PROC_MAPDEBUG
- DCMD_PROC_MAPDEBUG_BASE
- DCMD_PROC_SIGNAL
- DCMD_PROC_STATUS
- DCMD_PROC_TIDSTATUS
- DCMD_PROC_GETGREG

- DCMD_PROC_SETGREG
- DCMD_PROC_GETFPREG
- DCMD_PROC_SETFPREG
- DCMD_PROC_GETREGSET
- DCMD_PROC_SETREGSET
- DCMD_PROC_EVENT
- DCMD_PROC_SET_FLAG
- DCMD_PROC_CLEAR_FLAG
- DCMD_PROC_PAGEDATA
- DCMD_PROC_GETALTREG
- DCMD_PROC_SETALTREG
- DCMD_PROC_TIMERS
- DCMD_PROC_IRQS
- DCMD_PROC_THREADCTL
- DCMD_PROC_CHANNELS

Thread information

Several of the *devctl()* commands use a **procfs_status** structure (which is the same as **debug_thread_t**), so let's look at this structure before going into the commands themselves:

- DCMD_PROC_STATUS
- DCMD_PROC_STOP
- DCMD_PROC_TIDSTATUS
- DCMD_PROC_WAITSTOP

The **debug_thread_t** structure is defined as follows in **<sys/debug.h>**:

```
typedef struct _debug_thread_info {
    pid_t                       pid;
    pthread_t                   tid;
    _Uint32t                    flags;
    _Uint16t                    why;
    _Uint16t                    what;
    _Uint64t                    ip;
    _Uint64t                    sp;
    _Uint64t                    stkbase;
    _Uint64t                    tls;
    _Uint32t                    stksize;
```

```
    _Uint32t                    tid_flags;
    _Uint8t                     priority;
    _Uint8t                     real_priority;
    _Uint8t                     policy;
    _Uint8t                     state;
    _Int16t                     syscall;
    _Uint16t                    last_cpu;
    _Uint32t                    timeout;
    _Int32t                     last_chid;
    sigset_t                    sig_blocked;
    sigset_t                    sig_pending;
    siginfo_t                   info;
    union {
        struct {
            pthread_t                   tid;
        }                           join;
        struct {
            _Int32t                     id;
            _Uintptrt                   sync;
        }                           sync;
        struct {
            _Uint32t                    nd;
            pid_t                       pid;
            _Int32t                     coid;
            _Int32t                     chid;
            _Int32t                     scoid;
        }                           connect;
        struct {
            _Int32t                     chid;
        }                           channel;
        struct {
            pid_t                       pid;
            _Uintptrt                   vaddr;
            _Uint32t                    flags;
        }                           waitpage;
        struct {
            _Uint32t                    size;
        }                           stack;
        _Uint64t                        filler[4];
    }                           blocked;
    _Uint64t                    start_time;
    _Uint64t                    sutime;
    _Uint8t                     extsched[8];
    _Uint64t                    reserved2[5];
}                           debug_thread_t;
```

The members include:

pid, *tid*
: The process and thread IDs.

flags
: A combination of the following bits:

- _DEBUG_FLAG_STOPPED — the thread isn't running.
- _DEBUG_FLAG_ISTOP — the thread is stopped at a point of interest.
- _DEBUG_FLAG_IPINVAL — the instruction pointer isn't valid.
- _DEBUG_FLAG_ISSYS — system process.
- _DEBUG_FLAG_SSTEP — stopped because of single-stepping.

- _DEBUG_FLAG_CURTID — the thread is the current thread.
- _DEBUG_FLAG_TRACE_EXEC — stopped because of a breakpoint.
- _DEBUG_FLAG_TRACE_RD — stopped because of read access.
- _DEBUG_FLAG_TRACE_WR — stopped because of write access.
- _DEBUG_FLAG_TRACE_MODIFY — stopped because of modified memory.
- _DEBUG_FLAG_RLC — the Run-on-Last-Close flag is set.
- _DEBUG_FLAG_KLC — the Kill-on-Last-Close flag is set.
- _DEBUG_FLAG_FORK — the child inherits flags (stop on fork or spawn).

why One of the following:

- _DEBUG_WHY_REQUESTED
- _DEBUG_WHY_SIGNALLED
- _DEBUG_WHY_FAULTED
- _DEBUG_WHY_JOBCONTROL
- _DEBUG_WHY_TERMINATED
- _DEBUG_WHY_CHILD
- _DEBUG_WHY_EXEC

what The contents of this field depend on the *why* field:

why	***what***
_DEBUG_WHY_TERMINATED	The process's exit status
_DEBUG_WHY_SIGNALLED	*si_signo*
_DEBUG_WHY_FAULTED	*si_fltno*
_DEBUG_WHY_REQUESTED	0

ip The current instruction pointer.

sp The thread's stack pointer.

stkbase The base address of the thread's stack region.

tls A pointer to the **struct thread_local_storage *tls** (which will be on the thread's stack). For more information, see "Local storage for private data" in the entry for *ThreadCreate()* in the QNX Neutrino *Library Reference*.

stksize The stack size.

tid_flags — The thread flags; see _NTO_TF_* in **<sys/neutrino.h>**.

priority — The priority the thread is actually running at (e.g. its priority may have been boosted).

real_priority — The actual priority the thread would be at with no boosting and so on.

policy — The scheduling policy; one of SCHED_FIFO, SCHED_RR, SCHED_OTHER, or SCHED_SPORADIC.

state — The thread's state. The states themselves are defined in **<sys/states.h>**; for descriptions, see "Thread life cycle" in the QNX Neutrino Microkernel chapter of the *System Architecture* guide. If the thread is waiting for something, the *blocked* member may hold additional information, as described below.

syscall — The last system call; one of the __KER_* values defined in **<sys/kercalls.h>**.

last_cpu — The processor the thread last ran on.

timeout — **_NTO_TF_ACTIVE|_NTO_TF_IMMEDIATE|(1 << state)** — set by *TimerTimeout()*.

last_chid — The ID of the last channel this thread received a message on.

sig_blocked — The set of signals that are blocked for the thread.

sig_pending — The set of signals that are pending for the thread.

info — The **struct siginfo** of the last signal or fault received.

blocked — A **union** of the following:

- *join* — if the *state* is STATE_JOIN or STATE_WAITTHREAD, this structure contains *tid*, the ID of the thread that this thread is waiting for.
- *sync* — if the *state* is STATE_CONDVAR, STATE_MUTEX, or STATE_SEM, this structure contains:

  id — The address of the synchronization object.

  sync — For a condvar, this is a pointer to the associated mutex; for a mutex, it's a pointer to the mutex.
- *connect* — if the *state* is STATE_SEND or STATE_REPLY, this structure contains the node descriptor (*nd*), process ID (*pid*), connection ID (*coid*), channel ID (*chid*), and server connection ID (*scoid*) that the thread is waiting for.
- *channel* — if the *state* is STATE_RECEIVE, this structure contains *chid*, the ID of the channel that the thread is waiting for.

- *waitpage* — if the *state* is STATE_WAITPAGE, this structure contains:

  pid The ID of the process whose address space was active when the page fault occurred.

  vaddr The virtual address for which the thread is waiting for physical memory to be allocated.

  flags Internal use only.

- *stack* — if the *state* is STATE_STACK, this structure contains *size*, the amount of stack that the thread is waiting for to be allocated.

start_time The thread's starting time, in nanoseconds.

sutime The thread's system plus user running time, in nanoseconds.

extsched Extended scheduling information; a **struct extsched_aps_dbg_thread** structure if the adaptive partitioning thread scheduler is installed.

DCMD_PROC_BREAK

Set or remove a breakpoint in the process that's associated with the file descriptor. You must have opened the file descriptor for writing.

The argument is a pointer to a **procfs_break** structure (see **debug_break_t** in **<sys/debug.h>**) that specifies the breakpoint to be set or removed. For example:

```
procfs_break        brk;

memset(&brk, 0, sizeof brk);
brk.type = _DEBUG_BREAK_EXEC;
brk.addr = acc->break_addr.offset;
brk.size = 0;
devctl(fd, DCMD_PROC_BREAK, &brk, sizeof brk, 0);
```

Use a size of 0 to set a breakpoint, and a size of -1 to delete it.

Breakpoints other than _DEBUG_BREAK_EXEC are highly dependent on the hardware. In many architectures, other types of breakpoints cause the kernel to make the process run in single-step, checking the watchpoints each time, which can be very slow.

DCMD_PROC_CHANNELS

Get information about the channels owned by the specified process. Call this the first time with an argument of NULL to get the number of channels:

```
devctl(fd, DCMD_PROC_CHANNELS, NULL, 0, &n);
```

Next, allocate a buffer that's large enough to hold a **procfs_channel** structure (see **debug_channel_t** in **<sys/debug.h>**) for each channel, and pass it to another *devctl()* call:

```
my_buffer = (procfs_channel *)
               malloc( sizeof(procfs_channel) * n );
if ( my_buffer == NULL ) {
  /* Not enough memory. */
}

devctl( fd, DCMD_PROC_CHANNELS, my_buffer,
        sizeof(procfs_channel) * n, &n);
```

DCMD_PROC_CLEAR_FLAG

Clear specific debug flags with the values provided for the process associated with the file descriptor. The flags that can be cleared are described in **<sys/debug.h>**. The argument is a pointer to an unsigned integer that specifies the debug flags to clear. For example:

```
int flags;

flags = _DEBUG_FLAG_KLC;        /* Kill-on-Last-Close flag */
devctl( fd, DCMD_PROC_CLEAR_FLAG, &flags, sizeof(flags), NULL);
```

To set the flags, use DCMD_PROC_SET_FLAG.

DCMD_PROC_CURTHREAD

Switch to another thread. The argument to this command is a **pthread_t** value that specifies the thread that you want to be made the current thread. For example:

```
if ((err=devctl( fd, DCMD_PROC_CURTHREAD, &tid,
                 sizeof(tid), NULL)) != EOK) {
  /* An error occurred. */
}
```

DCMD_PROC_EVENT

Define an event to be delivered when the process associated with the file descriptor reaches a point of interest. Use the DCMD_PROC_RUN command to set up the point of interest.

The DCMD_PROC_EVENT command won't work unless you've set _DEBUG_RUN_ARM in the *flags* field of the **procfs_run** structure for the DCMD_PROC_RUN command.

Unlike DCMD_PROC_WAITSTOP, the DCMD_PROC_EVENT command doesn't block the calling process.

The argument is a pointer to the **sigevent** that you want to be delivered at the appropriate time. For example:

```
struct sigevent     event;

// Define a sigevent for process stopped notification.
event.sigev_notify = SIGEV_SIGNAL_THREAD;
event.sigev_signo = SIGUSR2;
event.sigev_code = 0;
event.sigev_value.sival_ptr = prp;
event.sigev_priority = -1;
devctl( fd, DCMD_PROC_EVENT, &event, sizeof(event), NULL);
```

DCMD_PROC_FREEZETHREAD

Freeze a thread in the process that's associated with the file descriptor. You must have opened the file descriptor for writing.

The argument is a pointer to a **pthread_t** value that specifies the thread to be frozen. For example:

```
devctl( fd, DCMD_PROC_FREEZETHREAD, &tid, sizeof tid, 0);
```

To unfreeze the thread, use DCMD_PROC_THAWTHREAD.

DCMD_PROC_GETALTREG

Get the information stored in the alternate register set for the process associated with the file descriptor. The argument is a pointer to a **procfs_fpreg** structure (see **debug_fpreg_t** in **<sys/debug.h>**) that's filled in with the required information on return. If you provide a non-NULL extra argument, it's filled with the actual size of the register set. For example:

```
procfs_fpreg reg;
int regsize;

devctl( fd, DCMD_PROC_GETALTREG, &reg, sizeof(reg), &regsize);
```

If the thread hasn't used the alternate register set (e.g. AltiVec registers), the read may fail.

To set the alternate register set, use DCMD_PROC_SETALTREG.

DCMD_PROC_GETFPREG

Get the information stored in the Floating Point Data registers for the process associated with the file descriptor. The argument is a pointer to a **procfs_fpreg** structure (see **debug_fpreg_t** in **<sys/debug.h>**) that's filled in with the required information on return. If you provide a non-NULL extra argument, it's filled with the size of the data. For example:

```
procfs_fpreg my_fpreg;

devctl( fd, DCMD_PROC_GETFPREG, my_fpreg, sizeof(procfs_fpreg),
        &size);
```


> If the thread hasn't used any floating-point arithmetic, the read may fail because an FPU context has not yet been allocated.

To set the Floating Point Data registers, use DCMD_PROC_SETFPREG.

DCMD_PROC_GETGREG

Get the information stored in the CPU registers based on the current thread of the process associated with the file descriptor. The argument is a pointer to a **procfs_greg** structure (see **debug_greg_t** in **<sys/debug.h>**) that's filled in with the required information on return. If you provide a non-NULL extra argument, it's filled with the size of the data. For example:

```
procfs_greg my_greg;

devctl( fd, DCMD_PROC_GETGREG, my_greg, sizeof(procfs_greg),
        &size);
```

To set the CPU registers, use DCMD_PROC_SETGREG.

DCMD_PROC_GETREGSET

Read the given register set. The argument is a pointer to a **procfs_regset** structure that's filled in with the required information on return. For example:

```
procfs_regset regset;

regset.id = REGSET_PERFREGS;
devctl( fd, DCMD_PROC_GETREGSET, &regset, sizeof(regset), NULL );
```

To set a given register set, use DCMD_PROC_SETREGSET.

DCMD_PROC_GET_BREAKLIST

Get a list of the active breakpoints for the process associated with the file descriptor. You must have opened the file descriptor for writing.

Call this the first time with an argument of NULL to get the number of breakpoints:

```
devctl( fd, DCMD_PROC_GET_BREAKLIST, NULL, 0, &n);
```

The total number of breakpoints returned is provided as the extra field. Next, allocate a buffer that's large enough to hold a **procfs_break** structure (see **debug_break_t** in **<sys/debug.h>**) for each breakpoint, and pass it to another *devctl()* call:

```
my_buffer = (procfs_break *) malloc( sizeof(procfs_break) * n );
if ( my_buffer == NULL ) {
  /* Not enough memory. */
}
devctl( fd, DCMD_PROC_GET_BREAKLIST, my_buffer,
        sizeof(procfs_break) * n, &n);
```

To set or clear breakpoints, use DCMD_PROC_BREAK.

DCMD_PROC_INFO

Obtain information about the process associated with the file descriptor. The argument is a pointer to a **procfs_info** structure (see **debug_process_t** in **<sys/debug.h>**) that's filled in with the required information on return. For example:

```
procfs_info my_info;

devctl( fd, DCMD_PROC_INFO, &my_info, sizeof(my_info), NULL);
```

DCMD_PROC_IRQS

Get the interrupt handlers owned by the process associated with the file descriptor. Call this the first time with an argument of NULL to get the number of interrupt handlers:

```
devctl( fd, DCMD_PROC_IRQS, NULL, 0, &n);
```

Next, allocate a buffer that's large enough to hold a **procfs_irq** structure (see **debug_irq_t** in **<sys/debug.h>**) for each handler, and pass it to another *devctl()* call:

```
my_buffer = (procfs_irq *) malloc( sizeof(procfs_irq) * n );
if ( my_buffer == NULL ) {
  /* Not enough memory. */
}

devctl( fd, DCMD_PROC_IRQS, my_buffer, sizeof(procfs_irq) * n,
        &n);
```

DCMD_PROC_MAPDEBUG

Get the best guess to the ELF object on the host machine. This is used by debuggers to find the object that contains the symbol information, even though it may have been stripped on the target machine. This call is useful only on MAP_ELF mappings. If any relocation of the ELF object was done, this translation will be undone. This lets you pass in an address within a ELF module, and get in return the address that the original object was linked at so a debugger can find the symbol. (This is an extension from the SYSV interface.)

The argument is a pointer to a **procfs_debuginfo** structure that's filled in with the required information on return. The **procfs_debuginfo** structure can specify the base address of the mapped segment that you're interested in. For example:

```
procfs_debuginfo map;

map.info.vaddr = some_vaddr;
devctl( fd, DCMD_PROC_MAPDEBUG, &map, sizeof map, NULL);
```

DCMD_PROC_MAPDEBUG is useful for non-ELF objects if you need to get the name. Note that the *path* member in **procfs_debuginfo** is a one-byte array; if you want to get the name, you need to allocate more space for it. For example:

```
struct {
    procfs_debuginfo   info;
    char               buff[_POSIX_PATH_MAX];
} map;
```

DCMD_PROC_MAPDEBUG_BASE

Get information pertaining to the path associated with the process associated with the file descriptor. This is a convenience extension; it's equivalent to using DCMD_PROC_INFO, and then DCMD_PROC_MAPDEBUG with the *base_address* field. The base address is the address of the initial executable.

The argument is a pointer to a **procfs_debuginfo** structure, which is filled in with the required information on return. For example:

```
procfs_debuginfo dinfop;

devctl( fd, DCMD_PROC_MAPDEBUG_BASE, &dinfop, sizeof(dinfop),
        NULL);
```

DCMD_PROC_MAPINFO

Obtain segment-specific information about mapped memory segments in the process associated with the file descriptor. This call matches the corresponding *mmap()* calls.

Individual page data isn't returned (i.e. the PG_* flags defined in **<mman.h>** aren't returned). If you need the page attributes, use DCMD_PROC_PAGEDATA instead.

Call this the first time with an argument of NULL to get the number of map entries:

```
devctl( fd, DCMD_PROC_MAPINFO, NULL, 0, &n);
```

Next, allocate a buffer that's large enough to hold a **procfs_mapinfo** structure for each map entry, and pass it to another *devctl()* call:

```
my_buffer = (procfs_mapinfo *)
               malloc( sizeof(procfs_mapinfo) * n );
if ( my_buffer == NULL ) {
  /* Not enough memory. */
}

devctl( fd, DCMD_PROC_MAPINFO, my_buffer,
        sizeof(procfs_mapinfo) * n, &n);
```

DCMD_PROC_PAGEDATA

Obtain page data about mapped memory segments in the process associated with the file descriptor. This call matches the corresponding *mmap()* calls.

If you need the segment-specific attributes, use DCMD_PROC_MAPINFO instead.

Call this the first time with an argument of NULL to get the number of map entries:

```
devctl(fd, DCMD_PROC_PAGEDATA, NULL, 0, &n);
```

Next, allocate a buffer that's large enough to hold a **procfs_mapinfo** structure for each map entry, and pass it to another *devctl()* call:

```
my_buffer = (procfs_mapinfo *)
                malloc( sizeof(procfs_mapinfo) * n );
if ( my_buffer == NULL ) {
  /* Not enough memory. */
}

devctl( fd, DCMD_PROC_PAGEDATA, my_buffer,
        sizeof(procfs_mapinfo) * n, &n);
```

DCMD_PROC_RUN

Resume the process that's associated with the file descriptor, if it has previously been stopped. You must have opened the file descriptor for writing. To stop the process, use DCMD_PROC_STOP.

The DCMD_PROC_RUN command also lets you set the "points of interest" (e.g. signals or faults you want to stop on) and other run flags (e.g. instruction pointer or single-step).

The argument is a pointer to a **procfs_run** structure (see **debug_run_t** in **<sys/debug.h>**). This structure is passed on as control information to the process before it resumes. For example:

```
procfs_run      run;

memset( &run, 0, sizeof(run) );
run.flags |= _DEBUG_RUN_CLRFLT | _DEBUG_RUN_CLRSIG;
devctl( fd, DCMD_PROC_RUN, &run, sizeof(run), 0);
```

The **procfs_run** or **debug_run_t** structure is defined as follows:

```
typedef struct _debug_run {
    _Uint32t                    flags;
    pthread_t                   tid;
    sigset_t                    trace;
    sigset_t                    hold;
    fltset_t                    fault;
    _Uintptrt                   ip;
} debug_run_t;
```

The members include:

flags
: A combination of zero or more of the following bits:

- _DEBUG_RUN_CLRSIG — clear pending signal.
- _DEBUG_RUN_CLRFLT — clear pending fault.
- _DEBUG_RUN_TRACE — the *trace* mask flags interesting signals.
- _DEBUG_RUN_FAULT — the *fault* mask flags interesting faults.
- _DEBUG_RUN_VADDR — change *ip* before running.
- _DEBUG_RUN_STEP — single-step only one thread.
- _DEBUG_RUN_STEP_ALL — single-step one thread; other threads run.
- _DEBUG_RUN_CURTID — change the current thread (target thread) to the one whose thread ID is specified by *tid*.
- _DEBUG_RUN_ARM — deliver an event at the point of interest. Use the DCMD_PROC_EVENT command to define the event.

tid
: The ID of the thread that you want to become the current thread, for use with _DEBUG_RUN_CURTID.

trace
: A set of signals (SIG*) to trace, for use with _DEBUG_RUN_TRACE.

hold
: Not currently used.

fault
: A set of faults (FLT*) to trace, for use with _DEBUG_RUN_FAULT.

ip
: The new value for the instruction pointer, for use with _DEBUG_RUN_VADDR.

Use *sigemptyset()* and *sigaddset()* to build the set of signals or faults for the *trace*, *hold* and *fault* members.

DCMD_PROC_SETALTREG

Set the alternate register set with the values provided for the process associated with the file descriptor. You must have opened the file descriptor for writing. The argument is a pointer to a **procfs_fpreg** structure (see **debug_fpreg_t** in **<sys/debug.h>**) that specifies to set the values of the alternate register set. For example:

```
procfs_fpreg reg;

/* Set the members of reg as required. */
devctl( fd, DCMD_PROC_SETALTREG, &reg, sizeof(reg), NULL);
```

To get the alternate register set, use DCMD_PROC_GETALTREG.

DCMD_PROC_SETFPREG

Set the Floating Point Data registers with the values provided for the process associated with the file descriptor. You must have opened the file descriptor for writing. The argument is a pointer to a **procfs_fpreg** structure (see **debug_fpreg_t** in **<sys/debug.h>**) that specifies the values of the Floating Point Data registers. For example:

```
procfs_fpreg my_fpreg;

/* Set the members of my_fpreg as required. */
devctl( fd, DCMD_PROC_SETFPREG, my_fpreg, sizeof(procfs_fpreg),
        NULL);
```

To get the Floating Point Data registers, use DCMD_PROC_GETFPREG.

DCMD_PROC_SETGREG

Set the CPU registers with the values provided for the process associated with the file descriptor. You must have opened the file descriptor for writing. The argument is a pointer to a **procfs_greg** structure (see **debug_greg_t** in **<sys/debug.h>**) that specifies the values to assign to the CPU registers. For example:

```
procfs_greg my_greg;

/* Set the members of my_greg as required. */
devctl( fd, DCMD_PROC_SETGREG, my_greg, sizeof(procfs_greg),
        NULL);
```

To get the CPU registers, use DCMD_PROC_GETGREG.

DCMD_PROC_SETREGSET

Set the given register set. The argument is a pointer to a **procfs_regset** structure that specifies the values to assign to the register set. For example:

```
procfs_regset regset;

regset.id = REGSET_PERFREGS;
devctl( fd, DCMD_PROC_SETREGSET, &regset, sizeof(regset), NULL );
```

To get the given register set, use DCMD_PROC_GETREGSET.

DCMD_PROC_SET_FLAG

Set specific debug flags with the values provided for the process associated with the file descriptor. The flags that can be set are described in **<sys/debug.h>**. The argument is a pointer to an unsigned integer that specifies the debug flags to set. For example:

```
int flags;

flags = _DEBUG_FLAG_KLC;        /* Kill-on-Last-Close flag */
devctl( fd, DCMD_PROC_SET_FLAG, &flags, sizeof(flags), NULL);
```

To clear the debug flags, use DCMD_PROC_CLEAR_FLAG.

DCMD_PROC_SIGNAL

Drop a signal on the process that's associated with the file descriptor. This is a way for a debugger to artificially generate signals as if they came from the system.

The argument is a pointer to a **procfs_signal** structure that specifies the signal to send. For example:

```
procfs_signal   signal;

signal.tid = 0;
signal.signo = SIGCONT;
signal.code = 0;
signal.value = 0;

devctl( fd, DCMD_PROC_SIGNAL, &signal, sizeof signal, NULL);
```

DCMD_PROC_STATUS

Get the current status of the current thread in the process associated with the file descriptor. The argument is a pointer to a **procfs_status** structure (see **debug_thread_t** in **<sys/debug.h>**) that's filled in with the required information on return. For example:

```
procfs_status my_status;

devctl( fd, DCMD_PROC_STATUS, &my_status, sizeof(my_status),
        NULL);
```

For more information about the contents of this structure, see "Thread information," earlier in this chapter.

DCMD_PROC_STOP

Stop the process that's associated with the file descriptor. You must have opened the file descriptor for writing.

The argument to this command is the address of a **procfs_status** structure (see **debug_thread_t** in **<sys/debug.h>**). This structure is filled with status information on return. For example:

```
procfs_status my_status;

devctl( fd, DCMD_PROC_STOP, &my_status, sizeof(my_status), NULL);
```

For more information about the contents of this structure, see "Thread information," earlier in this chapter.

To resume the process, use DCMD_PROC_RUN.

DCMD_PROC_SYSINFO

Obtain information stored in the system page. The argument is a pointer to a **procfs_sysinfo** structure that's filled in with the required information upon return. To get the whole system page, you have to make two calls: the first gets the size required:

```
devctl( fd, DCMD_PROC_SYSINFO, NULL, 0, &totalsize );
```

You then allocate a buffer of the required size and pass that buffer to the second call:

```
buffer = malloc( totalsize );
devctl( fd, DCMD_PROC_SYSINFO, buffer, totalsize, NULL );
```

The **procfs_sysinfo** structure is the same as the system page; for more information, see "Structure of the system page" in the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

DCMD_PROC_THAWTHREAD

Unfreeze a thread in the process that's associated with the file descriptor. You must have opened the file descriptor for writing.

The argument is a pointer to a **pthread_t** value that specifies the thread to be thawed. For example:

```
devctl( fd, DCMD_PROC_THAWTHREAD, &tid, sizeof tid, 0);
```

To freeze a thread, use DCMD_PROC_FREEZETHREAD.

DCMD_PROC_THREADCTL

Perform a *ThreadCtl()* on another process/thread. The argument is a pointer to a **procfs_threadctl** structure. For example:

```
procfs_threadctl    tctl;

tctl.tid = tid;
tctl.cmd = _NTO_TCTL_NAME;

tn = (struct _thread_name *)(&tctl.data);
tn->name_buf_len = sizeof(tctl.data) - sizeof(*tn);

//We can only communicate a maximum buffer size via devctl
if( newname_len > tn->name_buf_len || prevname_len >
    tn->name_buf_len) {
    return E2BIG;
}

tn->new_name_len = newname_len;
if(newname_len > 0) {
    memcpy(tn->name_buf, newname, newname_len);
}

devctl(fd, DCMD_PROC_THREADCTL, &tctl, sizeof(tctl), NULL);
```

DCMD_PROC_TIDSTATUS

Get the current status of a thread in the process associated with the file descriptor. This is a short form of using DCMD_PROC_CURTHREAD to set the current thread, then DCMD_PROC_STATUS to get information about that thread, and then restoring the current thread.

The argument is a pointer to a **procfs_status** structure (see **debug_thread_t** in **<sys/debug.h>**), with the required thread ID specified in the *tid* field. This structure is filled in with the required information on return. For example:

```
procfs_status my_status;

my_status.tid = 1;
devctl( fd, DCMD_PROC_TIDSTATUS, &my_status, sizeof(my_status),
        NULL);
```

For more information about the contents of this structure, see "Thread information," earlier in this chapter.

DCMD_PROC_TIMERS

Get the timers owned by the process associated with the file descriptor. Call this the first time with an argument of NULL to get the number of timers:

```
devctl( fd, DCMD_PROC_TIMERS, NULL, 0, &n);
```

Next, allocate a buffer that's large enough to hold a **procfs_timer** structure (see **debug_timer_t** in **<sys/debug.h>**) for each timer, and pass it to another *devctl()* call:

```
my_buffer = (procfs_timer *) malloc( sizeof(procfs_timer) * n;
if ( my_buffer == NULL ) {
  /* Not enough memory. */
}

devctl( fd, DCMD_PROC_TIMERS, my_buffer,
        sizeof(procfs_timer) * n, &n);
```

DCMD_PROC_WAITSTOP

Hold off the calling process until the process that's associated with the file descriptor reaches a point of interest. You must have opened the file descriptor for writing. Use the DCMD_PROC_RUN command to set up the point of interest. If you don't want to block the calling process, use DCMD_PROC_EVENT instead of DCMD_PROC_WAITSTOP.

The argument is a pointer to a **procfs_status** structure (see **debug_thread_t** in **<sys/debug.h>**) that's filled with status information on return. For example:

```
procfs_status my_status;

devctl( fd, DCMD_PROC_WAITSTOP, &my_status, sizeof my_status, 0);
```

For more information about the contents of this structure, see “Thread information,” earlier in this chapter.

Chapter 4

Tick, Tock: Understanding the Neutrino Microkernel's Concept of Time

In this chapter...

What's a tick?

When you're dealing with timing, every moment within the Neutrino microkernel is referred to as a *tick*. A tick is measured in milliseconds; its initial length is determined by the clock rate of your processor:

- If your CPU is 40 MHz or better, a tick is 1 ms.
- For slower processors, a tick represents 10 ms.

Programmatically you can change the clock period via the *ClockPeriod()* function.

Oversleeping: errors in delays

The tick size becomes important just about every time you ask the kernel to do something relating to pausing or delaying your process. This includes calls to the following functions:

- *select()*
- *alarm()*
- *nanosleep()*
- *nanospin()*
- *delay()*
- the whole family of *timer_*()* functions.

Normally, you use these functions assuming they'll do exactly what you say: "Sleep for 8 seconds!", "Sleep for 1 minute!", and so on. Unfortunately, you get into problems when you say "Sleep for 1 millisecond, ten thousand times!"

Delaying for a second: inaccurate code

Does this code work assuming a 1 ms tick?

```
void OneSecondPause() {

  /* Wait 1000 milliseconds. */
  for ( i=0; i < 1000; i++ ) delay(1);
}
```

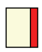
Unfortunately, no, this won't return after one second on IBM PC hardware. It'll likely wait for three seconds. In fact, when you call any function based on the *nanosleep()* or *select()* functions, with an argument of *n* milliseconds, it actually takes anywhere from *n* to infinity milliseconds. But more than likely, this example will take three seconds.

So why exactly does this function take three seconds?

Timer quantization error

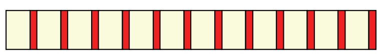
What you're seeing is called *timer quantization error*. One aspect of this error is actually something that's so well understood and accepted that it's even documented in a standard: the POSIX Realtime Extension (1003.1b-1993/1003.1i-1995). This document says that it's all right to delay too much, but it *isn't* all right to delay too little — the premature firing of a timer is undesirable.

Since the calling of *delay()* is asynchronous with the running of the clock interrupt, the kernel has to add one clock tick to a relative delay to ensure the correct amount of time (consider what would happen if it didn't, and a one-tick delay was requested just before the clock interrupt went off).



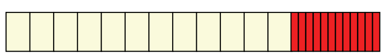
A single 1 ms sleep with error.

That normally adds half a millisecond each time, but in the example given, you end up synchronized with the clock interrupt, so the full millisecond gets tacked on each time.



Twelve 1 ms sleeps with each one's error.

The small error on each sleep accumulates:



Twelve 1 ms sleeps with the accumulated error.

OK, that should make the loop last 2 seconds — where's the extra second coming from?

The tick and the hardware timer

The problem is that when you request a 1 ms tick rate, the kernel may not be able to actually give it to you because of the frequency of the input clock to the timer hardware. In such cases, it chooses the closest number that's faster than what you requested. In terms of IBM PC hardware, requesting a 1 ms tick rate actually gets you 999,847 nanoseconds between each tick. With the requested delay, that gives us the following:

- 1,000,000 ns + 999,847 ns = 1,999,847 ns of actual delay
- 1,999,847 ns / 999,847 ns = 2.000153 ticks before the timer expires

Since the kernel expires timers only at a clock interrupt, the timer expires after **`ceil(2.000153)`** ticks, so each **`delay(1)`** call actually waits:

999,847 ns * 3 = 2,999,541 ns

Multiply that by a 1000 for the loop count, and you get a total loop time of 2.999541 seconds.

Delaying for a second: better code

So this code should work?

```
void OneSecondPause() {

    /* Wait 1000 milliseconds. */
   for ( i=0; i < 100; i++ ) delay(10);
}
```

It will certainly get you closer to the time you expect, with an accumulated error of only 1/10 of a second.

Another hiccup with hardware timers

The hardware timer of the PC has another side effect when it comes to dealing with timers. The "Oversleeping: errors in delays" section above explains the behavior of the sleep-related functions. Timers are similarly affected by the design of the PC hardware.

For example, let's consider the following C code:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <sys/syspage.h>

int main( int argc, char *argv[] )
{
    int pid;
    int chid;
    int pulse_id;
    timer_t timer_id;
    struct sigevent event;
    struct itimerspec timer;
    struct _clockperiod clkper;
    struct _pulse pulse;
    _uint64 last_cycles=-1;
    _uint64 current_cycles;
    float cpu_freq;
    time_t start;

    /* Get the CPU frequency in order to do precise time
       calculations. */
    cpu_freq =  SYSPAGE_ENTRY( qtime )->cycles_per_sec;

    /* Set our priority to the maximum, so we won't get disrupted
       by anything other than interrupts. */
    {
        struct sched_param param;
        int ret;
```

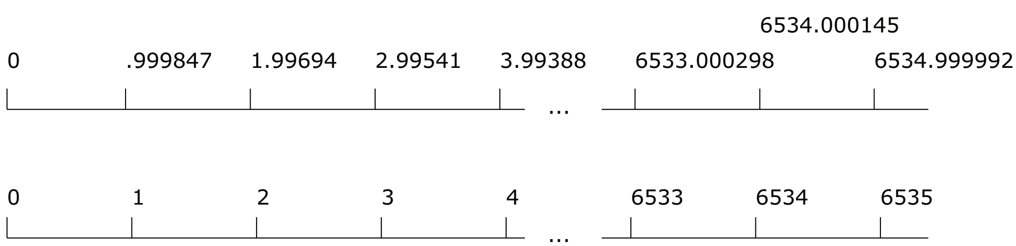
```
    param.sched_priority = sched_get_priority_max( SCHED_RR );
    ret = sched_setscheduler( 0, SCHED_RR, &param);
    assert ( ret != -1 );
}

/* Create a channel to receive timer events on. */
chid = ChannelCreate( 0 );
assert ( chid != -1 );

/* Set up the timer and timer event. */
event.sigev_notify            = SIGEV_PULSE;
event.sigev_coid              = ConnectAttach ( ND_LOCAL_NODE,
                                                0, chid, 0, 0 );
event.sigev_priority          = getprio(0);
event.sigev_code              = 1023;
event.sigev_value.sival_ptr  = (void*)pulse_id;

assert ( event.sigev_coid != -1 );

if ( timer_create(  CLOCK_REALTIME, &event, &timer_id ) == -1 )
{
    perror ( "can't create timer" );
    exit( EXIT_FAILURE );
}

/* Change the timer request to alter the behavior. */
#if 1
      timer.it_value.tv_sec        = 0;
      timer.it_value.tv_nsec       = 1000000;
      timer.it_interval.tv_sec     = 0;
      timer.it_interval.tv_nsec    = 1000000;
#else
      timer.it_value.tv_sec        = 0;
      timer.it_value.tv_nsec       = 999847;
      timer.it_interval.tv_sec     = 0;
      timer.it_interval.tv_nsec    = 999847;
#endif

/* Start the timer. */
if ( timer_settime( timer_id, 0, &timer, NULL ) == -1 )
{
    perror("Can't start timer.\n");
    exit( EXIT_FAILURE );
}

/* Set the tick to 1 ms. Otherwise if left to the default of
   10 ms, it would take 65 seconds to demonstrate. */
clkper.nsec       = 1000000;
clkper.fract      = 0;
ClockPeriod ( CLOCK_REALTIME, &clkper, NULL, 0  );   // 1ms

/* Keep track of time. */
start = time(NULL);
for( ;; )
{
    /* Wait for a pulse. */
    pid = MsgReceivePulse ( chid, &pulse, sizeof( pulse ),
                            NULL );

    /* Should put pulse validation here... */
    current_cycles = ClockCycles();

    /* Don't print the first iteration. */
```

```
        if ( last_cycles != -1 )
        {
            float elapse = (current_cycles - last_cycles) /
                           cpu_freq;

            /* Print a line if the request is 1.05 ms longer than
               requested. */
            if ( elapse > .00105 )
            {
                 printf("A lapse of %f ms occurred at %d seconds\n",
                         elapse, time( NULL ) - start );
            }
        }

        last_cycles = current_cycles;
    }
}
```

The program checks to see if the time between two timer events is greater than 1.05 ms. Most people expect that given QNX Neutrino's great realtime behavior, such a condition will never occur, but it will, not because the kernel is misbehaving, but because of the limitation in the PC hardware. It's impossible for the OS to generate a timer event at exactly 1.0 ms; it will be .99847 ms. This has unexpected side effects.

Where's the catch?

As described earlier in this chapter, there's a 153-nanosecond (ns) discrepancy between the request and what the hardware can do. The kernel timer manager is invoked every .999847 ms. Every time a timer fires, the kernel checks to see if the timer is periodic and, if so, adds the number of nanoseconds to the expected timer expiring point, no matter what the current time is. This phenomenon is illustrated in the following diagram:



Actual and expected timer expirations.

The first line illustrates the actual time at which timer management occurs. The second line is the time at which the kernel expects the timer to be fired. Note what happens at 6534: the next value appears not to have incremented by 1 ms, thus the event 6535 *won't* be fired!

For signal frequencies, this phenomenon is called a *beat*. When two signals of various frequencies are "added," a third frequency is generated. You can see this effect if you use your camcorder to record a TV image. Because a TV is updated at 60 Hz, and camcorders usually operate on a different frequency, at playback, you can often see a white line that scrolls in the TV image. The speed of that line is related to the difference in frequency between the camcorder and the TV.

In this case we have two frequencies, one at 1000 Hz, and the other at 1005.495 Hz. Thus, the beat frequency is 1.5 micro Hz, or one blip every 6535 milliseconds.

This behavior has the benefit of giving you the expected number of fired timers, on average. In the example above, after 1 minute, the program would have received 60000 fired timer events (1000 events /sec * 60 sec). If your design requires very precise timing, you have no other choice but to request a timer event of .999847 ms and not 1 ms. This can make the difference between a robot moving very smoothly or scratching your car.

Chapter 5

Transparent Distributed Processing Using Qnet

In this chapter. . .

Transparent Distributed Processing (TDP) allows you to leverage the processing power of your entire network by sharing resources and services transparently over the network. TDP uses Neutrino native network protocol Qnet to link the devices in your network.

What is Qnet?

Qnet is Neutrino's protocol for distributed networking. Using Qnet, you can build a transparent distributed-processing platform that is fast and scalable. This is accomplished by extending the Neutrino message passing architecture over a network. This creates a group of tightly integrated Neutrino nodes (systems) or CPUs — a Neutrino native network.

A program running on a Neutrino node in this Qnet network can transparently access any resource, whether it's a file, device, or another process. These resources reside on any other node (a computer, a workstation or a CPU in a system) in the Qnet network. The Qnet protocol builds an optimized network that provides a fast and seamless interface between Neutrino nodes.

For a high-level description, see Native Networking (Qnet) in the *System Architecture* guide; for information about what the *user* needs to know about networking, see Using Qnet for Transparent Distributed Processing in the Neutrino *User's Guide*.

For more advanced topics and programming hints on Qnet, see Advanced Qnet Topics appendix.

Benefits of Qnet

The Qnet protocol extends interprocess communication (IPC) transparently over a network of microkernels. This is done by taking advantage of the Neutrino's message-passing paradigm. Message passing is the central theme of Neutrino that manages a group of cooperating processes by routing messages. This enhances the efficiency of all transactions among all processes throughout the system.

For more information about message passing and Qnet, see Advanced Qnet Topics appendix.

What works best

The Qnet protocol is deployed as a network of trusted machines. It lets these machines share all their resources efficiently with minimum overhead. This is accomplished by allowing a client process to send a message to a remote manager in the same way that it sends a message to a local one. See the "How does it work?" section of this chapter. For example, using Qnet, you can use the Neutrino utilities (**cp**, **mv** and so on) to manipulate files anywhere on the Qnet Network as if they were on your machine — by communicating with the filesystem manager on the remote nodes. In addition, the Qnet protocol doesn't do any authentication of remote requests. Files are protected by the normal permissions that apply to users and groups (see "File ownership and permissions" in Working with Files in the *User's Guide*).

Qnet, through its distributed processing platform, lets you do the following tasks efficiently:

- access your remote filesystem
- scale your application with unprecedented ease
- write applications using a collection of cooperating processes that communicate transparently with each other using Neutrino message passing
- extend your application easily beyond a single processor or symmetric multi-processor to several single processor machines and distribute your processes among these processors
- divide your large application into several processes that coordinate their work using messages
- debug your application easily for processes that communicate at a very low level, and that use Neutrino's memory protection feature
- use builtin remote procedure call functionality

Since Qnet extends Neutrino message passing over the network, other forms of interprocess communication (e.g. signals, message queues, and named semaphores) also work over the network.

What type of application is well-suited for Qnet?

Any application that inherently needs more than one computer, due to its processing or physical layout requirements, could likely benefit from Qnet.

For example, you can apply Qnet networking successfully in many industrial-automation applications (e.g. a fabrication plant, with computers scattered around). From an application standpoint, Qnet provides an efficient form of distributed computing where all computers look like one big computer because Qnet extends the fundamental Neutrino message passing across all the computers.

Another useful application is in the telecom space, where you need to implement large routers that have several processors. From an architectural standpoint, these routers generally have some interface cards and a central processor that runs a set of server processes. Each interface card, in turn, has a processor that runs another set of interface (e.g. client) processes. These client processes communicate via Qnet using Neutrino message passing with the server processes on the central processor, as if they were all running on the same processor. The scalability of Qnet allows more and more interface cards to be plugged into the router, without any code changes required to the application.

Qnet drivers

In order to support different hardware, you may need to write a driver for Qnet. The driver essentially performs three functions: transmits a packet, receives a packet, and resolves the remote node's interface.

In most cases, you don't need a specific driver for your hardware, for example, for implementing a local area network using Ethernet hardware or for implementing TCP/IP networking that require IP encapsulation. In these cases, the underlying **io-pkt*** and TCP/IP layer is sufficient to interface with the Qnet layer for transmitting and receiving packets. You use standard Neutrino drivers to implement Qnet over a local area network or to encapsulate Qnet messages in IP (TCP/IP) to allow Qnet to be routed to remote networks.

But suppose you want to set up a very tightly coupled network between two CPUs over a super-fast interconnect (e.g. PCI or RapidIO). You can easily take advantage of the performance of such a high-speed link, because Qnet can talk directly to your hardware driver. There's no **io-pkt*** layer in this case. All you need is a little code at the very bottom of Qnet layer that understands how to transmit and receive packets. This is simple as there is a standard internal API between the rest of Qnet and this very bottom portion, the driver interface. Qnet already supports different packet transmit/receive interfaces, so adding another is reasonably straightforward. The transport mechanism of Qnet (called the *L4*) is quite generic and can be configured for different size MTUs, whether or not ACK packets or CRC checks are required, to take the full advantage of your link's advanced features (e.g. guaranteed reliability).

For details about how to write a driver, see the section on "Writing a driver for Qnet" later in this chapter.

The source for Transparent Distributed Processing is available from the Foundry 27 part of our website. It will help you develop custom drivers and/or modify Qnet components to suit your particular application.

How does it work?

As explained in the *System Architecture* guide, Neutrino client and server applications communicate by Neutrino message passing. Function calls that need to communicate with a manager application, such as the POSIX functions *open()*, *write()*, *read()*, *ioctl()*, or other functions such as *devctl()* are all built on Neutrino message passing.

Qnet allows these messages to be sent over a network. If these messages are being sent over a network, how is a message sent to a remote manager vs a local manager?

When you access local devices or manager processes (such as a serial device, TCP/IP socket, or **mqueue**), you access these devices by opening a pathname under **/dev**. This may be apparent in the application source code:

```
/*Open a serial device*/
fd = open("/dev/ser1",O_RDWR....);
```

or it may not. For example, when you open a socket:

```
/*Create a UDP socket*/
sock = socket(AF_INET, SOCK_DGRAM, 0);
```

The *socket()* function opens a pathname under **/dev** called **/dev/socket/2** (in the case of AF_INET, which is address family two). The *socket()* function call uses this pathname to establish a connection with the socket manager (**io-pkt***), just as the *open()* call above established a connection to the serial device manager (**devc-ser8250**).

The magic of this is that you access all managers by the name that they added to the pathname space. For more information, see the *Writing a Resource Manager* guide.

When you enable the Qnet native network protocol, the pathname spaces of all the nodes in your Qnet network are added to yours. The pathname space of remote nodes appears (by default) under the prefix **/net**.

Under QNX 4, you use a double slash followed by a node number to refer to another node.

The **/net** directory is created by the Qnet protocol manager (**lsm-qnet.so**). If, for example, the other node is called **node1**, its pathname space appears as follows:

```
/net/node1/dev/socket
/net/node1/dev/ser1
/net/node1/home
/net/node1/bin
....
```

So with Qnet, you can now open pathnames (files or managers) on other remote Qnet nodes, in the same way that you open files locally. This means that you can access regular files or manager processes on other Qnet nodes as if they were executing on your local node.

First, let's see some basic examples of Qnet use:

- To display the contents of a file on another machine (**node1**), you can use **less**, specifying the path through **/net**:

  ```
  less /net/node1/etc/TIMEZONE
  ```

- To get system information about all of the remote nodes that are listed in **/net**, use **pidin** with the **net** argument:

  ```
  $ pidin net
  ```

- You can use **pidin** with the **-n** option to get information about the processes on another machine:

  ```
  pidin -n node1 | less
  ```

- You can even run a process on another machine, using the **-f** option to the **on** command:

  ```
  on -f node date
  ```

In all of these uses, the application source or the libraries (for example **libc**) they depend on, simply open the pathnames under **/net**. For example, if you wish to make use of a serial device on another node **node1**, perform an *open()* function with the pathname **/net/node1/dev/ser1** i.e.

```
fd = open("/net/node1/dev/ser1",O_RDWR...);
```

As you can see, the code required for accessing remote resources and local resources is identical. The only change is the pathname used.

In the TCP/IP *socket()* case, it's the same, but implemented differently. In the socket case, you don't directly open a filename. This is done inside the socket library. In this case, an environment variable is provided to set the pathname for the socket call (the **SOCK** environment variable — see **io-pkt***).

Some other applications are:

Remote filesystem access
: In order to access **/tmp/file1** file on **node1** remotely from another node, use **/net/node1/tmp/file1** in *open()*.

Message queue
: You can create or open a message queue by using *mq_open()*. The **mqueue** manager must be running. When a queue is created, it appears in the pathname space under **/dev/mqueue**. So, you can access **/dev/mqueue** on **node1** from another node by using **/net/node1/dev/mqueue**.

The alternate implementation of message queues that uses the **mq** server and asynchronous messages doesn't support access to a queue via Qnet.

Semaphores
: Using Qnet, you can create or access named semaphores in another node. For example, use **/net/node1/semaphore_location** in the *sem_open()* function. This creates or accesses the named semaphore in **node1**.

This brings up an important issue for the client application or libraries that a client application uses. If you think that your application will be distributed over a network, you will want to include the capability to specify another pathname for connecting to your services. This way, your application will have the flexibility of being able to connect to local or remote services via a user-configuration adjustment. This could be as simple as the ability to pass a node name. In your code, you would add the prefix **/net/***node_name* to any pathname that may be opened on the remote node. In the local case, or default case if appropriate, you could omit this prefix when accessing local managers.

In this example, you're using standard resource managers, such as would be developed using the resource manager framework (see the *Writing a Resource Manager* guide). For further information, or for a more in-depth view of Qnet, see Advanced Qnet Topics appendix.

There is another design issue to contend with at this point: the above design is a static one. If you have services at known locations, or the user will be placing services at known locations, then this may be sufficient. It would be convenient, though, if your client application could locate these services automatically, without the need to know what nodes exist in the Qnet network, or what pathname they've added to the namespace. You can now use the Global Name Service (**gns**) manager to locate services with an arbitrary name representing that service. For example, you can locate a service with a name such as **printer** instead of opening a pathname of **/net/node/dev/par1** for a parallel port device. The **printer** name locates the parallel port manager process, whether it's running locally or remotely.

Locating services using GNS

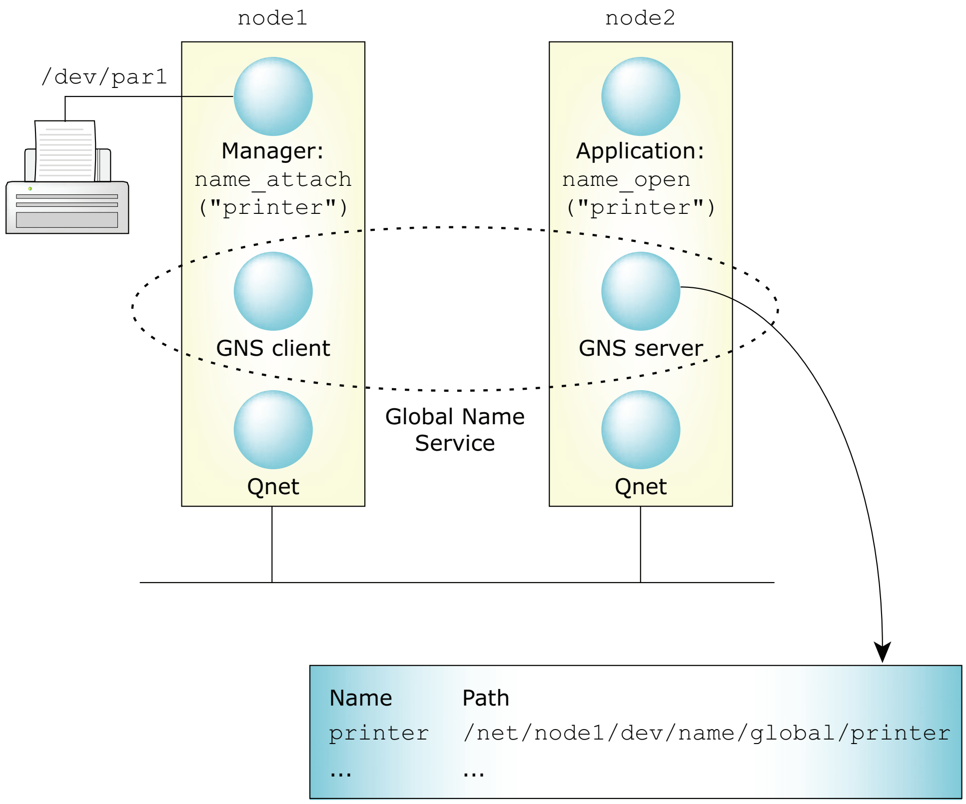
You use **gns**, the Global Name Service or GNS manager to locate services. GNS is a standalone resource manager. With the help of this utility, an application can advertise, look up, and use (connect to) a service across Qnet network, without knowing the details of where the service is, or who the provider is.

Different modes of gns

The **gns** utility runs in two different modes: server- and client-mode. A server-mode manager is a central database that stores advertised services, and handles lookup and connect requests. A client-mode manager relays advertisement, lookup, and connect requests between local application and the GNS server(s).

For more information on starting and configuring GNS, see the **gns** utility in the *Utilities Reference*.

Here's a simple layout for a GNS client and a GNS server distributed over a network:



A simple GNS setup.

In this example, there's one **gns** client and one **gns** server. As far as an application is concerned, the GNS service is one entity. The client-server relationship is only between **gns** processes (we'll examine this later). The server GNS process keeps track of the globally registered services, while the client GNS process on the other node relays **gns** requests for that node to the **gns** server.

When a client and server application interacts with the GNS service, they use the following APIs:

Server

name_attach() Register your service with the GNS server.

name_detach() Deregister your service with the GNS server.

Client

name_open() Open a service via the GNS server.

name_close() Close the service opened with *name_open()*.

Registering a service

In order to use GNS, you need to first register the manager process with GNS, by calling *name_attach()*.

When you register a service, you need to decide whether to register this manager's service locally or globally. If you register your service locally, only the local node is able to see this service; another node is not able to see it. This allows you to have client applications that look for service *names* rather than pathnames on the node it is executing on. This document highlights registering services globally.

When you register GNS service globally, any node on the network running a client application can use this service, provided the node is running a **gns** client process and is connected to the **gns** server, along with client applications on the nodes running the **gns** server process. You can use a typical *name_attach()* call as follows:

```
if ((attach = name_attach(NULL, "printer", NAME_FLAG_ATTACH_GLOBAL)) == NULL) {
     return EXIT_FAILURE;
     }
```

First thing you do is to pass the flag NAME_FLAG_ATTACH_GLOBAL. This causes your service to be registered globally instead locally.

The last thing to note is the *name*. This is the name that clients search for. This name can have a single level, as above, or it can be nested, such as **printer/ps**. The call looks like this:

```
if ((attach = name_attach(NULL, "printer/ps", NAME_FLAG_ATTACH_GLOBAL)) == NULL) {
     return EXIT_FAILURE;
     }
```

Nested names have no impact on how the service works. The only difference is how the services are organized in the filesystem generated by **gns**. For example:

```
$ ls -l /dev/name/global/
total 2
dr-xr-xr-x  0 root      techies           1 Feb 06 16:20 net
dr-xr-xr-x  0 root      techies           1 Feb 06 16:21 printer

$ ls -l /dev/name/global/printer
total 1
dr-xr-xr-x  0 root      techies           1 Feb 06 16:21 ps
```

The first argument to the *name_attach()* function is the dispatch handle. You pass a dispatch handle to *name_attach()* once you've already created a dispatch structure. If this argument is NULL, a dispatch structure is created automatically.

What happens if more than one instance of the server application (or two or more applications that register the same service name) are started and registered with GNS? This is treated as a redundant service. If one application terminates or detaches its service, the other service takes over. However, it's not a round-robin configuration; all requests go to one application until it's no longer available. At that point, the requests resolve to another application that had registered the same service. There is no guaranteed ordering.

There's no credential restriction for applications that are attached as local services. An application can attach a service globally only if the application has **root** privilege.

When your application is to terminate, or you wish not to provide access to the service via GNS, you should call *name_detach()*. This removes the service from GNS.

For more information, see *name_attach()* and *name_detach()*.

Your client should call *name_open()* to locate the service. If you wish to locate a global service, you need to pass the flag NAME_FLAG_ATTACH_GLOBAL:

```
if ((fd = name_open("printer", NAME_FLAG_ATTACH_GLOBAL)) == -1) {
     return EXIT_FAILURE;
     }
```

or:

```
if ((fd = name_open("printer/ps", NAME_FLAG_ATTACH_GLOBAL)) == -1) {
     return EXIT_FAILURE;
     }
```

If you don't specify this flag, GNS looks only for a local service. The function returns an **fd** that you can then use to access the service manager by sending messages, just as if you it had opened the service directly as **/dev/par1**, or **/net/***node***/dev/par1**.

GNS path namespace

A service is represented by a path namespace (without a leading "/") and is registered under **/dev/name/global** or **/dev/name/local**, depending on how it attaches itself. Every machine running a **gns** client or server on the same network has the same view of the **/dev/name/global** namespace. Each machine has its own local namespace **/dev/name/local** that reflects its own local services.

Here's an example after a service called **printer** has attached itself globally:

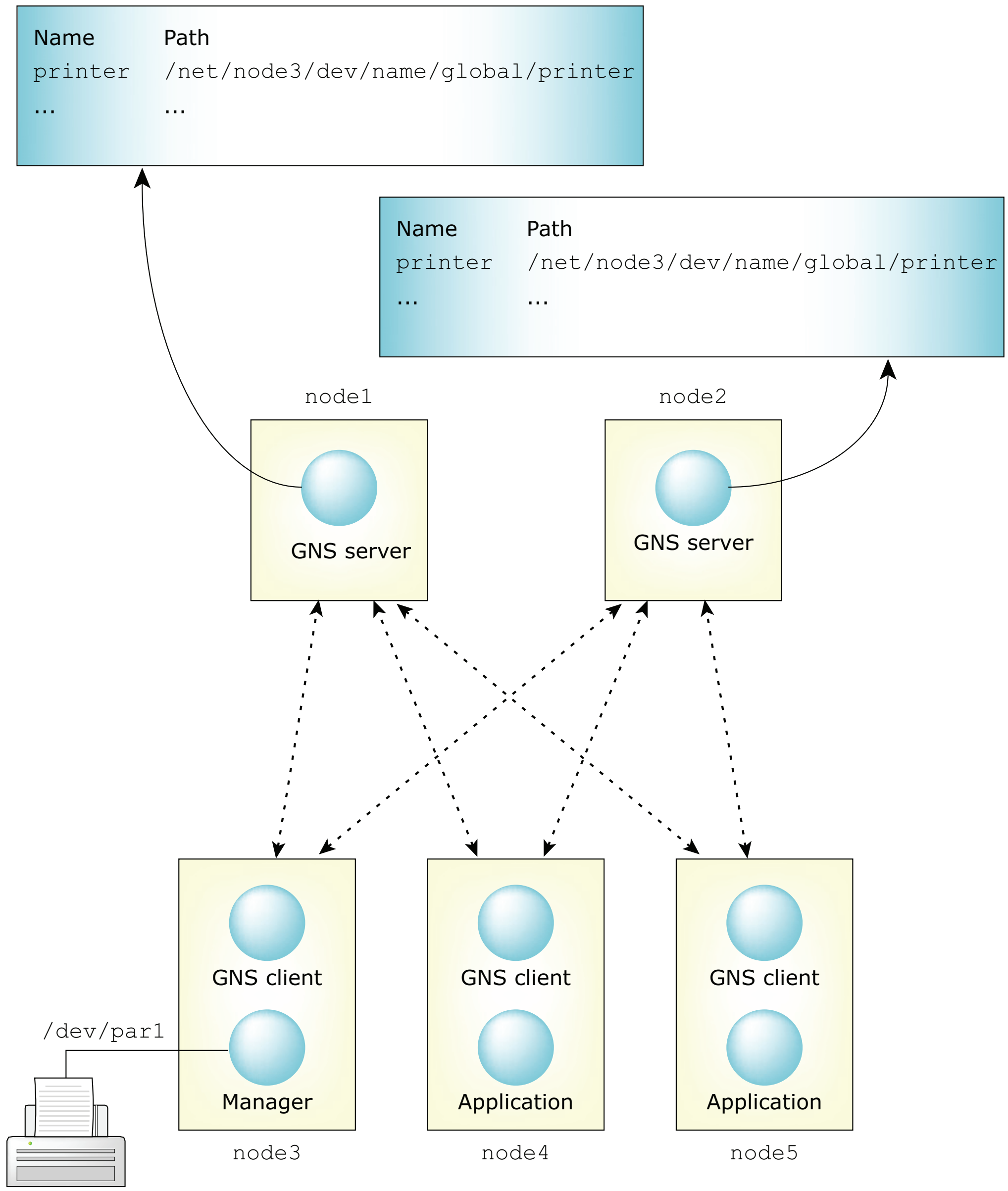
```
$ ls -l /dev/name/global/
total 2
dr-xr-xr-x  0 root      techies     1 Feb 06 16:20 net
dr-xr-xr-x  0 root      techies     1 Feb 06 16:21 printer
```

Deploying the gns processes

When you deploy the **gns** processes on your network, you start the **gns** process in two modes: server and client. You need at least one **gns** process running as a server on one node, and you can have one or more **gns** clients running on the remaining nodes. The role of the **gns** server process is to maintain the database that stores the advertised services. The role of a client **gns** process is to relay requests from its node to the **gns** server process on the other node. A **gns** process must be running on each node that wishes to access GNS.

It's possible to start multiple global name service managers (**gns** process) in server mode on different nodes. You can deploy server-mode **gns** processes in two ways: as redundant servers, or as servers that handle two or more different global domains.

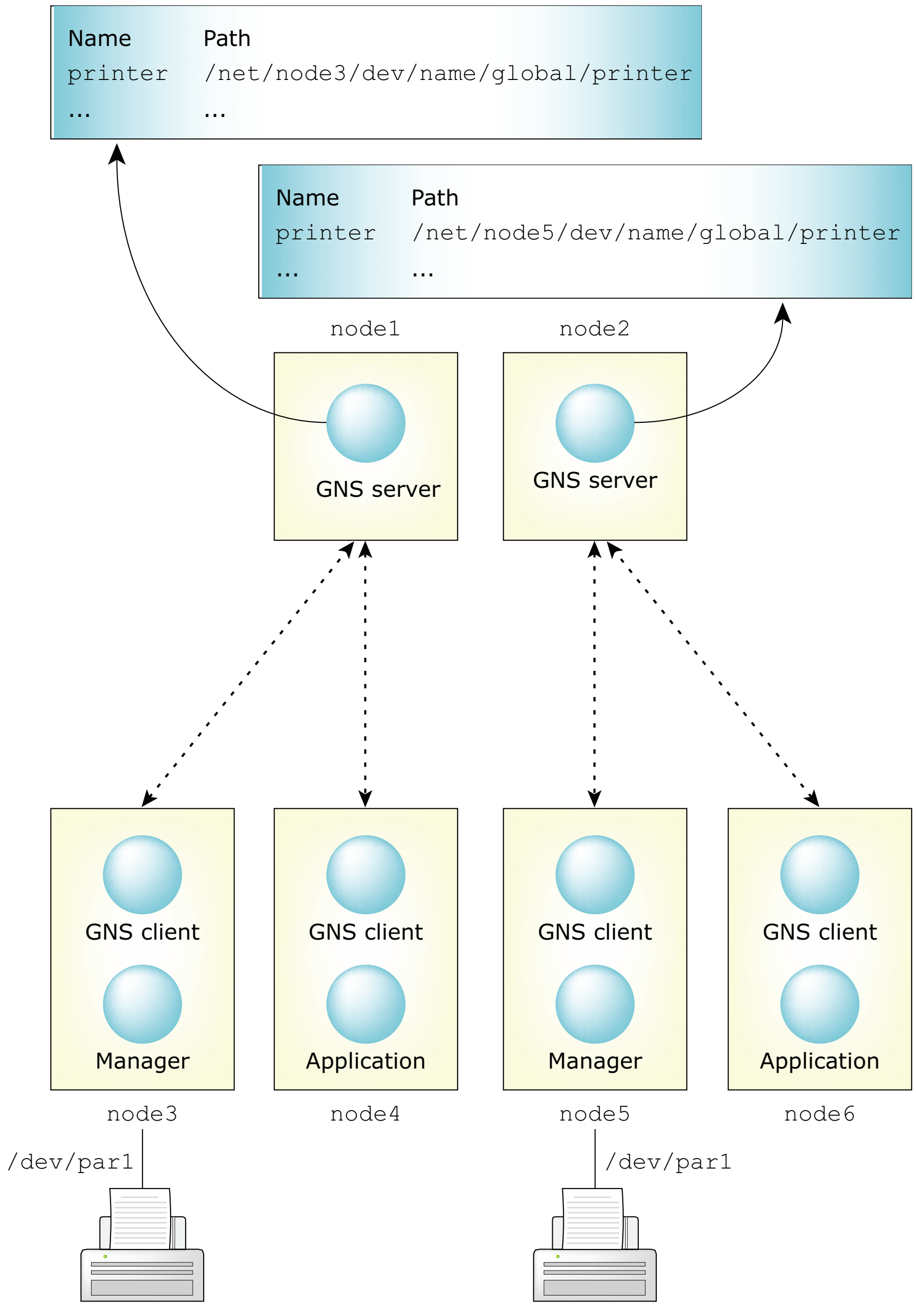
In the first scenario, you have two or more servers with identical database information. The **gns** client processes are started with contact information for both servers. Operations are then sent to all **gns** server processes. The **gns** servers, however, don't communicate with each other. This means that if an application on one **gns** server node wants to register a global service, another **gns** server can't do it. This doesn't affect other applications on the network, because when they connect to that service, both GNS servers are contacted.



A redundant GNS setup.

You don't have to start all redundant **gns** servers at the same time. You can start one **gns** server process first, and then start a second **gns** server process at a later time. In this case, use the special option **-s** *backup_server* on the second **gns** server process to make it download the current service database from another node that's already running the **gns** server process. When you do this, the clients connected to the first node (that's already running the **gns** server process) are notified of the existence of the other server.

In the second scenario, you maintain more than one global domain. For example, assume you have two nodes, each running a **gns** server process. You also have a client node that's running a **gns** client process and is connecting to one of the servers. A different client node connects to the other server. Each server node has unique services registered by each client. A client connected to server **node1** can't see the service registered on the server **node2**.



Separate global domains.

What is demonstrated in each scenario is that it's the client that determines whether a server is acting as a redundant server or not. If a client is configured to connect to two or more servers, then those servers are redundant servers for that client's services. The client can see the services that exist on those servers, and it registers its services with those servers.

There's no limit to the number of server mode **gns** processes that can be run on the network. Increasing the number of servers, however, in a redundant environment can increase network use and make **gns** function calls such as *name_attach()* more expensive as clients send requests to each server that exists in its configuration. It's recommended that you run only as many **gns** servers in a redundant configuration as your system design requires and no more than that.

For more information, see **gns** documentation in the *Utilities Reference*.

Quality of Service (QoS) and multiple paths

Quality of Service (QoS) is an issue that often arises in high-availability networks as well as realtime control systems. In the Qnet context, QoS really boils down to *transmission media selection* — in a system with two or more network interfaces, Qnet chooses which one to use, according to the policy you specify.

If you have only a single network interface, the QoS policies don't apply at all.

QoS policies

Qnet supports transmission over *multiple networks* and provides the following policies for specifying how Qnet should select a network interface for transmission:

loadbalance (the default)
: Qnet is free to use all available network links, and shares transmission equally among them.

preferred
: Qnet uses one specified link, ignoring all other networks (unless the preferred one fails).

exclusive
: Qnet uses one — and only one — link, ignoring all others, even if the exclusive link fails.

loadbalance

Qnet decides which links to use for sending packets, depending on current load and link speeds as determined by **io-pkt***. A packet is queued on the link that can deliver the packet the soonest to the remote end. This effectively provides greater bandwidth between nodes when the links are up (the bandwidth is the sum of the bandwidths of all available links) and allows a graceful degradation of service when links fail.

If a link does fail, Qnet switches to the next available link. By default, this switch takes a few seconds *the first time*, because the network driver on the bad link will have timed out, retried, and finally died. But once Qnet "knows" that a link is down, it will *not* send user data over that link. (This is a significant improvement over the QNX 4 implementation.)

The time required to switch to another link can be set to whatever is appropriate for your application using command-line options of Qnet. See **lsm-qnet.so** documentation.

Using these options, you can create a redundant behavior by minimizing the latency that occurs when switching to another interface in case one of the interfaces fail.

While load-balancing among the live links, Qnet sends periodic maintenance packets on the failed link in order to detect recovery. When the link recovers, Qnet places it back into the pool of available links.

The **loadbalance** QoS policy is the default.

preferred

With this policy, you specify a preferred link to use for transmissions. Qnet uses only that one link until it fails. If your preferred link fails, Qnet then turns to the other available links and resumes transmission, using the **loadbalance** policy.

Once your preferred link is available again, Qnet again uses only that link, ignoring all others (unless the preferred link fails).

exclusive

You use this policy when you want to lock transmissions to only one link. Regardless of how many other links are available, Qnet will latch onto the one interface you specify. And if that exclusive link fails, Qnet will *not* use any other link.

Why would you want to use the **exclusive** policy? Suppose you have two networks, one much faster than the other, and you have an application that moves large amounts of data. You might want to restrict transmissions to only the fast network, in order to avoid swamping the slow network if the fast one fails.

Specifying QoS policies

You specify the QoS policy as part of the pathname. For example, to access **/net/node1/dev/ser1** with a QoS of **exclusive**, you could use the following pathname:

```
/net/node1~exclusive:en0/dev/ser1
```

The QoS parameter always begins with a tilde (**~**) character. Here we're telling Qnet to lock onto the **en**0 interface exclusively, even if it fails.

Symbolic links

You can set up symbolic links to the various "QoS-qualified" pathnames:

```
ln -sP /net/note1~preferred:en1 /remote/sql_server
```

This assigns an "abstracted" name of **/remote/sql_server** to the node **node1** with a preferred QoS (i.e. over the **en**1 link).

You can't create symbolic links inside **/net** because Qnet takes over that namespace.

Abstracting the pathnames by one level of indirection gives you multiple servers available in a network, all providing the same service. When one server fails, the abstract pathname can be "remapped" to point to the pathname of a different server. For example, if **node1** fails, then a monitoring program could detect this and effectively issue:

```
rm /remote/sql_server
ln -sP /net/magenta /remote/sql_server
```

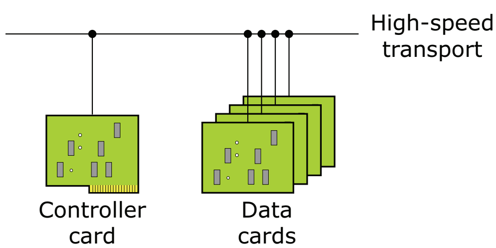
This removes **node1** and reassigns the service to **node2**. The real advantage here is that applications can be coded based on the abstract "service name" rather than be bound to a specific node name.

For a real world example of choosing appropriate QoS policy in an application, see the following section on designing a system using Qnet.

Designing a system using Qnet

The product

In order to explain the design of a system that takes advantage of the power of Qnet by performing distributed processing, consider a multiprocessor hardware configuration that is suitable for a typical telecom box. This configuration has a generic controller card and several data cards to start with. These cards are interconnected by a high-speed transport (HST) bus. The controller card configures the box by communicating with the data cards, and establishes/enables data transport in and out of the box (i.e. data cards) by routing packets.



The typical challenges to consider for this type of box include:

- Configuring the data cards
- Configuring the controller card
- Replacing a data card
- Enhancing reliability via multiple transport buses
- Enhancing reliability via multiple controller cards

Developing your distributed system

You need several pieces of software components (along with the hardware) to build your distributed system. Before going into further details, you may review the following sections from Using Qnet for Transparent Distributed Processing chapter in the Neutrino *User's Guide*:

- Software components for Qnet networking
- Starting Qnet
- Conventions for naming nodes

Configuring the data cards

Power up the data cards to start **procnto** and **qnet** in sequence. These data cards need a minimal amount of flash memory (e.g. typically 1 MB) to store the Neutrino image.

In the buildfile of the data cards, you should link the directories of the data cards to the controller cards as follows:

```
[type=link] /bin  = /net/cc0/bin
[type=link] /sbin = /net/cc0/sbin
[type=link] /usr  = /net/cc0/usr
```

where **cc0** is the name of the the controller card.

Assuming that the data card has a console and shell prompt, try the following commands:

```
$ ls /net
```

You get a list of boards running Neutrino and Qnet:

```
 cc0  dc0  dc1  dc2  dc3
```

Or, use the following command on a data card:

```
$ ls /net/cc0
```

You get the following output (i.e. the contents of the root of the filesystem for the controller card):

```
.                  .inodes           mnt0         tmp
..                 .longfilenames    mnt1         usr
.altboot           bin               net          var
.bad_blks          dev               proc         xfer
.bitmap            etc               sbin
.boot              home              scratch
```

Configuring the controller card

Configure the controller card in order to access different servers running on it — either by the data cards, or by the controller card itself. Make sure that the controller card has a larger amount of flash memory than the data cards do. This flash memory contains all the binaries, data and configuration files that the applications on the data cards access as if they were on a local storage device.

Call the following API to communicate with the **mqueue** server by any application:

```
mq_open("/net/cc0/dev/mqueue/app_q", ....)
```

A simple variation of the above command requires that you run the following command during initialization:

```
$ ln -s /net/cc0/dev/mqueue /mq
```

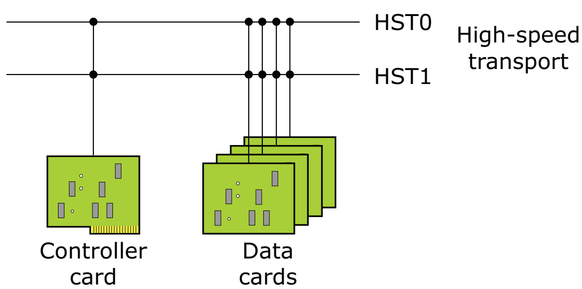
Then all applications, whether they're running on the data cards or on the controller card, can call:

```
mq_open("/mq/app_q", ....)
```

Similarly, applications can even utilize the TCP/IP stack running on the controller card.

Enhancing reliability via multiple transport buses

Qnet provides design choices to improve the reliability of a high-speed transport bus, most often a single-point of failure in such type of telecom box.



You can choose between different transport selections to achieve a different Quality of Service (or QoS), such as:

- load-balance — no interface specified
- preferred — specify an interface, but allow failover
- exclusive — specify an interface, no failover

These selections allow you to control how data will flow via different transports.

In order to do that, first, find out what interfaces are available. Use the following command at the prompt of any card:

```
ls /dev/io-net
```

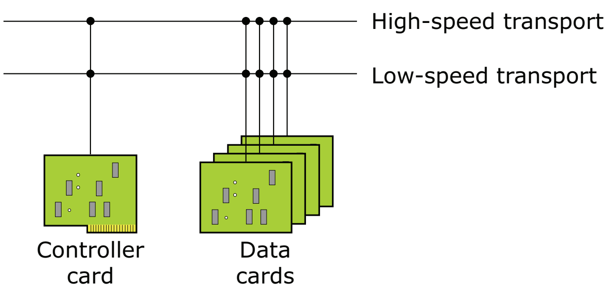
You see the following:
hs0 hs1

These are the interfaces available: HST 0 and HST 1.

Select your choice of transport as follows:

Use this command:	To select this transport:
`ls /net/cc0`	Loadbalance, the default choice
`ls /net/cc0~preferred:hs0`	Preferred. Try HST 0 first; if that fails, then transmit on HST 1.
`ls /net/cc0~exclusive:hs0`	Exclusive. Try HST 0 first. If that fails, terminate transmission.

You can have another economical variation of the above hardware configuration:



This configuration has asymmetric transport: a High-Speed Transport (HST) and a reliable and economical Low-Speed Transport (LST). You might use the HST for user data, and the LST exclusively for out-of-band control (which can be very helpful for diagnosis and during booting). For example, if you use generic Ethernet as the LST, you could use a **bootp** ROM on the data cards to economically boot — no flash would be required on the data cards.

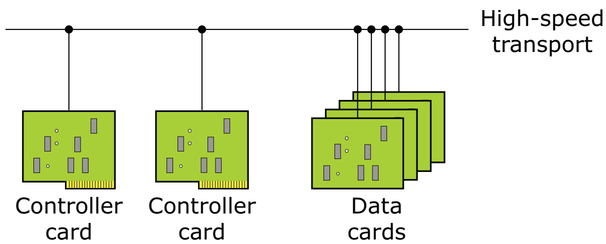
With asymmetric transport, use of the QoS policy as described above likely becomes even more useful. You might want some applications to use the HST link first, but use the LST if the HST fails. You might want applications that transfer large amounts of data to exclusively use the HST, to avoid swamping the LST.

Redundancy and scalability using multiple controller cards

Redundancy

The reliability of such a telecom box also hinges on the controller card, that's a critical component and certainly a potential SPOF (single point of failure). You can increase the reliability of this telecom box by using additional controller cards.

The additional controller card is for redundancy. Add another controller card as shown below:



Once the (second) controller card is installed, the challenge is in the determination of the primary controller card. This is done by the software running on the controller cards. By default, applications on the data cards access the primary controller card. Assuming **cc0** is the primary controller card, Use the following command to access this card in **/cc** directory:

```
ln -s /net/cc0 /cc
```

The above indirection makes communication between data card and controller card transparent. In fact, the data cards remain unaware of the number of controller cards, or which card is the primary controller card.

Applications on the data cards access the primary controller card. In the event of failure of the primary controller card, the secondary controller card takes over. The applications on the data cards redirect their communications via Qnet to the secondary controller card.

Scalability

You can also scale your resources to run a particular server application using additional controller cards. For example, if your controller card (either a SMP or non-SMP board) doesn't have the necessary resources (e.g. CPU cycle, memory), you could increase the total processor and box resources by using additional controller cards. Qnet transparently distributes the (load of) application servers across two or more controller cards.

Autodiscovery vs static

When you're creating a network of Neutrino hosts via Qnet, one thing you must consider is how they locate and address each other. This falls into two categories: autodiscovery and static mappings.

The decision to use one or the other can depend on security and ease of use.

The autodiscovery mechanism (i.e. **en_ionet**; see **lsm-qnet.so** for more information) allows Qnet nodes to discover each other automatically on a transport that supports broadcast. This is a very convenient and dynamic way to build your network, and doesn't require user intervention to access a new node.

One issue to consider is whether or not the physical link being used by your Qnet nodes is secure. Can another untrusted Qnet node be added to this physical network of Qnet nodes? If the answer is yes, you should consider another resolver (**file: filename**). If you use this resolver, only the nodes listed in the file can be accessed.

This file consists of node names and a string representing the addressing scheme of your transport layer. In the Ethernet case, this is the unique MAC address of the Qnet node listed. If you're using the file resolver for this purpose, you also want to specify the option **auto_add**=0 in **lsm-qnet.so**. This keeps your node from responding to node discovery protocol requests and adding a host that isn't listed in your resolver **file**.

Another available resolver, **dns** lets you access another Qnet node if you know its name (**IP**). This is used in combination with the IP transport (**lsm-qnet.so** option **bind=ip**). Since it doesn't have an **auto_add** feature as the **en_ionet** resolver does, you don't need to specify a similar Qnet option. Your Qnet node resolve the remote Qnet node's name only via the file used by the Qnet **file** resolver.

When should you use Qnet, TCP/IP, or NFS?

In your network design, when should you use Qnet, TCP/IP, or NFS? The decision depends on what your intended application is and what machines you need to connect.

The advantage of using Qnet is that it lets you build a truly distributed processing system with incredible scalability. For many applications, it could be a benefit to be able to share resources among your application systems (nodes). Qnet implements a native network protocol to build this distributed processing system.

The basic purpose of Qnet is to extend Neutrino message passing to work over a network link. It lets these machines share all their resources with little overhead. A Qnet network is a trusted environment where resources are tightly integrated, and remote manager processes can be accessed transparently. For example, with Qnet, you can use the Neutrino utilities (**cp**, **mv** and so on) to manipulate files anywhere on the Qnet network as if they were on your machine. Because it's meant for a group of trusted machines (such as you'd find in an embedded system), Qnet doesn't do any authentication of remote requests. Also, the application really doesn't know whether it's accessing a resource on a remote system; and most importantly, the application doesn't need any special code to handle this capability.

If you're developing a system that requires remote procedure calling (RPC), or remote file access, Qnet provides this capability transparently. In fact, you use a form of remote procedure call (a Neutrino message pass) every time you access a manager on your Neutrino system. Since Qnet creates an environment where there's no difference between accessing a manager locally or remotely, remote procedure calling (capability) is builtin. You don't need to write source code to distribute your services. Also, since you are sharing the filesystem between systems, there's no need for NFS to access files on other Neutrino hosts (of the same endian), because you can access remote filesystem managers the same way you access your local one. Files are protected by the normal permissions that apply to users and groups (see "File ownership and permissions" in the Working with Files chapter in the *User's Guide*).

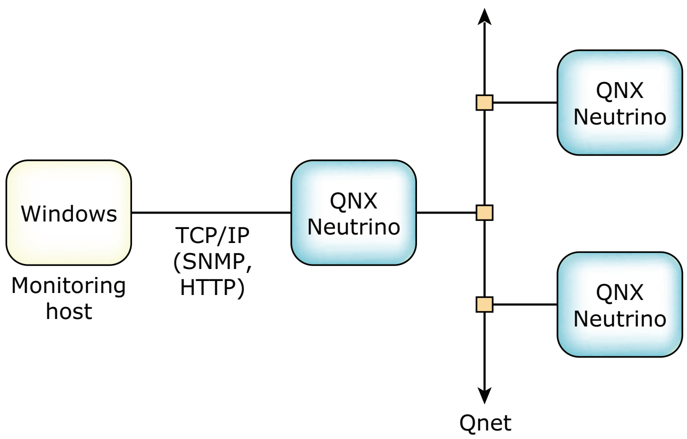
There are several ways to control access to a Qnet node, if required:

- Bind Qnet to a specific network interface; this ensures that the protocol functions only on that specific interface.

- Use **maproot** and **mapany** options to control — in a limited way — what other users can do on your system.
- Use a static list of your peer systems instead of dynamically discovering them.

You can also configure Qnet to be used on a local LAN, or routed over to a WAN if necessary (encapsulated in the IP protocol).

Depending on your system design, you may need to include TCP/IP protocols along with Qnet, or instead of Qnet. For example, you could use a TCP/IP-based protocol to connect your Qnet cluster to a host that's running another operating system, such as a monitoring station that controls your system, or another host providing remote access to your system. You'll probably want to deploy standard protocols (e.g SNMP, HTTP, or a **telnet** console) for this purpose. If all the hosts in your system are running different operating systems, then your likely choice to connect them would be TCP/IP. The TCP/IP protocols typically do authentication to control access; it's useful for connecting machines that you don't necessarily trust.



You can also build a Neutrino-based TCP/IP network. A Neutrino TCP/IP network can access resources located on any other system that supports TCP/IP protocol. For a discussion of Neutrino TCP/IP specifics, see TCP/IP Networking in the *System Architecture* guide.

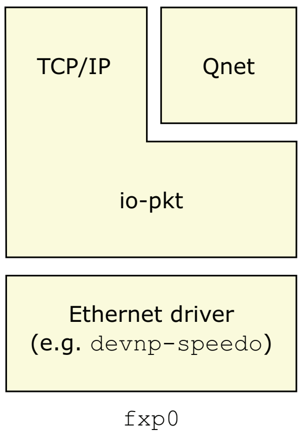
Another issue may be the required behavior. For example, NFS has been designed for filesystem operations between all hosts and all endians. It's widely supported and a connectionless protocol. In NFS, the server can be shut down and restarted, and the client resumes automatically. NFS also uses authentication and controls directory access. However, NFS retries forever to reach a remote host if it doesn't respond, whereas Qnet can return an error if connectivity is lost to a remote host. For more information, see "NFS filesystem" in Working with Filesystems in the *User's Guide*).

If you require broadcast or multicast services, you need to look at TCP/IP functionalities, because Qnet is based on Neutrino message passing, and has no concept of broadcasting or multicasting.

Writing a driver for Qnet

In order to support different hardware, you may need to write a driver for Neutrino's Qnet. The driver essentially performs three functions: transmitting a packet, receiving a packet, and resolving the remote node's interface (address). This section describes some of the issues you'll face when you need to write a driver.

First, let's define what exactly a driver is, from Qnet's perspective. When Qnet is run with its default binding of raw Ethernet (e.g. **bind=en**0), you'll find the following arrangement of layers that exists in the node:

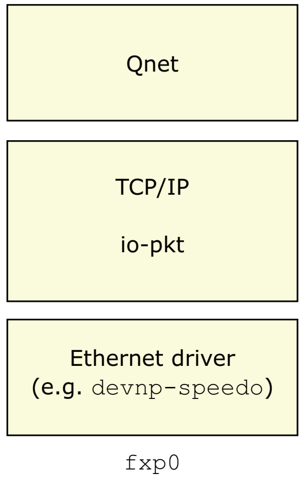


In the above case, **io-pkt*** is actually the **driver** that transmits and receives packets, and thus acts as a hardware-abstraction layer. Qnet doesn't care about the details of the Ethernet hardware or driver.

So, if you simply want new Ethernet hardware supported, you don't need to write a Qnet-specific driver. What you need is just a normal Ethernet driver that knows how to interface to **io-pkt***.

There is a bit of code at the very bottom of Qnet that's specific to **io-pkt*** and has knowledge of exactly how **io-pkt*** likes to transmit and receive packets. This is the L4 driver API abstraction layer.

Let's take a look at the arrangement of layers that exist in the node when Qnet is run with the optional binding of IP encapsulation (e.g. **bind=ip**):

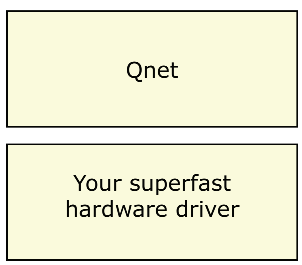


As far as Qnet is concerned, the TCP/IP stack is now its **`driver`**. This stack is responsible for transmitting and receiving packets.

Therefore, if IP encapsulation is acceptable for your application, you really don't need to write a Qnet **`driver`**, you can use any existing IP transport mechanism.

Again, it's worth mentioning that at the very bottom of Qnet there is a bit of code (L4 driver API) that's specific to TCP/IP and knows exactly how to transmit and receive packets using the TCP/IP stack.

If you have some superfast network hardware that you don't want to write an **`io-pkt*`** driver for, you could get the ultimate in performance by writing a dedicated driver. A possible arrangement of layers is as follows:



Just as before, Qnet needs a little code at the very bottom that knows exactly how to transmit and receive packets to this new driver. There exists a standard internal API (L4 driver API) between the rest of Qnet and this very bottom portion, the driver interface. Qnet already supports different packet transmit/receive interfaces, so adding another is reasonably straightforward. The transport mechanism of Qnet (called the L4) is quite generic, and can be configured for different size MTUs, whether or not ACK packets or CRC checks are required, to take the full advantage of your link's advanced features (e.g. guaranteed reliability).

Chapter 6

Writing an Interrupt Handler

In this chapter. . .

What's an interrupt?

The key to handling hardware events in a timely manner is for the hardware to generate an *interrupt*. An interrupt is simply a pause in, or interruption of, whatever the processor was doing, along with a request to do something else.

The hardware generates an interrupt whenever it has reached some state where software intervention is desired. Instead of having the software continually poll the hardware — which wastes CPU time — an interrupt is the preferred method of "finding out" that the hardware requires some kind of service. The software that handles the interrupt is therefore typically called an *Interrupt Service Routine* (ISR).

Although crucial in a realtime system, interrupt handling has unfortunately been a very difficult and awkward task in many traditional operating systems. Not so with Neutrino. As you'll see in this chapter, handling interrupts is almost trivial; given the fast context-switch times in Neutrino, most if not all of the "work" (usually done by the ISR) is actually done by a thread.

Let's take a look at the Neutrino interrupt functions and at some ways of dealing with interrupts. For a different look at interrupts, see the Interrupts chapter of *Getting Started with QNX Neutrino*.

Interrupts on multicore systems

On a multicore system, each interrupt is directed to one (and only one) CPU, although it doesn't matter which. How this happens is under control of the programmable interrupt controller chip(s) on the board. When you initialize the PICs in your system's startup, you can program them to deliver the interrupts to whichever CPU you want to; on some PICs you can even get the interrupt to rotate between the CPUs each time it goes off.

For the startups we write, we typically program things so that all interrupts (aside from the one(s) used for interprocessor interrupts) are sent to CPU 0. This lets us use the same startup for both **procnto** and **procnto-smp**. According to a study that Sun did a number of years ago, it's more efficient to direct all interrupts to one CPU, since you get better cache utilization.

For more information, see the Customizing Image Startup Programs chapter of *Building Embedded Systems*.

An ISR (Interrupt Service Routine) that's added by *InterruptAttach()* runs on the CPU that takes the interrupt.

An IST (Interrupt Service Thread) that receives the event set up by *InterruptAttachEvent()* runs on any CPU, limited only by the scheduler and the runmask.

A thread that calls *InterruptWait()* runs on any CPU, limited only by the scheduler and the runmask.

Attaching and detaching interrupts

In order to install an ISR, the software must tell the OS that it wishes to associate the ISR with a particular source of interrupts, which can be a hardware *Interrupt Request* line (IRQ) or one of several software interrupts. The actual number of interrupts depends on the hardware configuration supplied by the board's manufacturer. For the interrupt assignments for specific boards, see the sample build files in **${**QNX_TARGET**}/${**PROCESSOR**}/boot/build**.

In any case, a thread specifies which interrupt source it wants to associate with which ISR, using the *InterruptAttach()* or *InterruptAttachEvent()* function calls; when the software wishes to dissociate the ISR from the interrupt source, it can call *InterruptDetach()*. For example:

```
#define IRQ3 3

/*  A forward reference for the handler */
extern const sigevent *serint (void *, int);
...

/*
 *  Associate the interrupt handler, serint,
 *  with IRQ 3, the 2nd PC serial port
 */
ThreadCtl( _NTO_TCTL_IO, 0 );
id = InterruptAttach (IRQ3, serint, NULL, 0, 0);
...

/*  Perform some processing. */
...

/*  Done; detach the interrupt source. */
InterruptDetach (id);
```

The startup code is responsible for making sure that all interrupt sources are masked during system initialization. When the first call to *InterruptAttach()* or *InterruptAttachEvent()* is done for an interrupt vector, the kernel unmasks it. Similarly, when the last *InterruptDetach()* is done for an interrupt vector, the kernel remasks the level.

Because the interrupt handler can potentially gain control of the machine, we don't let just anybody associate an interrupt.

The thread must have *I/O privileges* — the privileges associated with being able to manipulate hardware I/O ports and affect the processor interrupt enable flag (the x86 processor instructions **in**, **ins**, **out**, **outs**, **cli**, and **sti**). Since currently only the **root** account can gain I/O privileges, this effectively limits the association of interrupt sources with ISR code.

Let's now take a look at the ISR itself.

Interrupt Service Routine (ISR)

In our example above, the function *serint()* is the ISR. In general, an ISR is responsible for:

- determining which hardware device requires servicing, if any
- performing some kind of servicing of that hardware (usually this is done by simply reading and/or writing the hardware's registers)
- updating some data structures shared between the ISR and some of the threads running in the application
- signalling the application that some kind of event has occurred

Depending on the complexity of the hardware device, the ISR, and the application, some of the above steps may be omitted.

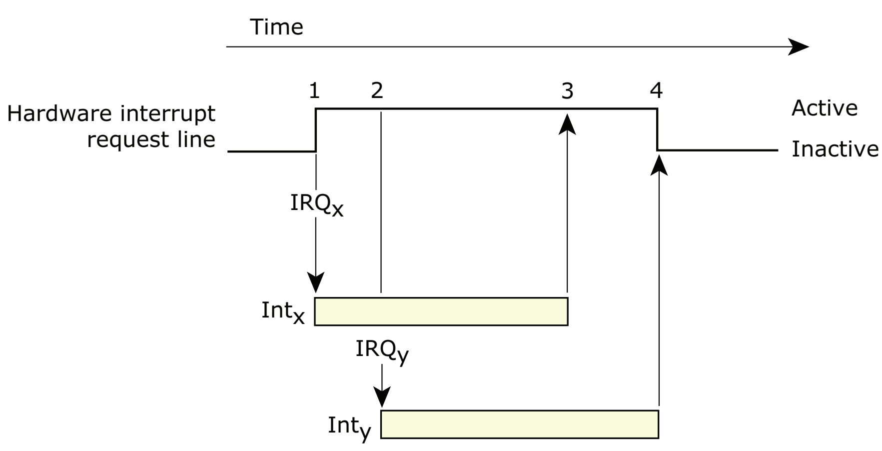
Let's take a look at these steps in turn.

Determining the source of the interrupt

Depending on your hardware configuration, there may actually be *multiple* hardware sources associated with an interrupt. This issue is a function of your specific hardware and bus type. This characteristic (plus good programming style) mandates that your ISR ensure that the hardware associated with it actually *caused* the interrupt.

Most *PIC* (Programmable Interrupt Controller) chips can be programmed to respond to interrupts in either an *edge-sensitive* or *level-sensitive* manner. Depending on this programming, interrupts may be sharable.

For example:



Interrupt request assertion with multiple interrupt sources.

In the above scenario, if the PIC is operating in a level-sensitive mode, the IRQ is considered active whenever it's high. In this configuration, while the second assertion (step 2) doesn't itself *cause* a new interrupt, the interrupt is still considered active even when the original cause of the interrupt is removed (step 3). Not until the last assertion is cleared (step 4) will the interrupt be considered inactive.

In edge-triggered mode, the interrupt is "noticed" only once, at step 1. Only when the interrupt line is cleared, and then reasserted, does the PIC consider another interrupt to have occurred.

Neutrino allows ISR handlers to be *stacked*, meaning that multiple ISRs can be associated with one particular IRQ. The impact of this is that each handler in the chain must look at its associated hardware and determine if it caused the interrupt. This works reliably in a level-sensitive environment, but not an edge-triggered environment.

To illustrate this, consider the case where two hardware devices are sharing an interrupt. We'll call these devices "HW-A" and "HW-B." Two ISR routines are attached to one interrupt source (via the *InterruptAttach()* or *InterruptAttachEvent()* call), in sequence (i.e. ISR-A is attached first in the chain, ISR-B second).

Now, suppose HW-B asserts the interrupt line first. Neutrino detects the interrupt and dispatches the two handlers in order — ISR-A runs first and decides (correctly) that its hardware did *not* cause the interrupt. Then ISR-B runs and decides (correctly) that its hardware *did* cause the interrupt; it then starts servicing the interrupt. But before ISR-B clears the source of the interrupt, suppose HW-A asserts an interrupt; what happens depends on the type of IRQ.

Edge-triggered IRQ

If you have an edge-triggered bus, when ISR-B clears the source of the interrupt, the IRQ line is still held active (by HW-A). But because it's edge-triggered, the PIC is waiting for the next clear/assert transition before it decides that another interrupt has occurred. Since ISR-A already ran, it can't possibly run again to actually *clear* the source of the interrupt. The result is a "hung" system, because the interrupt will *never* transit between clear and asserted again, so no further interrupts on that IRQ line will ever be recognized.

Level-sensitive IRQ

On a level-sensitive bus, when ISR-B clears the source of the interrupt, the IRQ line is still held active (by HW-A). When ISR-B finishes running and Neutrino sends an *EOI* (End Of Interrupt) command to the PIC, the PIC immediately reinterrupts the kernel, causing ISR-A (and then ISR-B) to run.

Since ISR-A clears the source of the interrupt (and ISR-B doesn't do anything, because its associated hardware doesn't require servicing), everything functions as expected.

Servicing the hardware

The above discussion may lead you to the conclusion that "level-sensitive is *good*; edge-triggered is *bad*." However, another issue comes into play.

In a level-sensitive environment, your ISR *must* clear the source of the interrupt (or at least mask it via *InterruptMask()*) before it completes. (If it didn't, then when the kernel issued the EOI to the PIC, the PIC would then immediately reissue a processor interrupt and the kernel would loop forever, continually calling your ISR code.)

In an edge-triggered environment, there's no such requirement, because the interrupt won't be noticed again until it transits from clear to asserted.

In general, to actually service the interrupt, your ISR has to do very little; the minimum it can get away with is to clear the source of the interrupt and then schedule a thread to actually do the work of handling the interrupt. This is the recommended approach, for a number of reasons:

- Context-switch times between the ISR completing and a thread executing are very small — typically on the order of a few microseconds.
- The type of functions that the ISR itself can execute is very limited (those that don't call any kernel functions, except the ones listed below).
- The ISR runs at a priority *higher* than any software priority in the system — having the ISR consume a significant amount of processor has a negative impact on the realtime aspects of Neutrino.

Since the range of hardware attached to an interrupt source can be very diverse, the specific how-to's of servicing the interrupt are beyond the scope of this document — this really depends on what your hardware requires you to do.

Safe functions

When the ISR is servicing the interrupt, it can't make any kernel calls (except for the few that we'll talk about shortly). This means that you need to be careful about the library functions that you call in an ISR, because their underlying implementation may use kernel calls.

For a list of the functions that you can call from an ISR, see the Summary of Safety Information appendix in the *Library Reference*.

Here are the only kernel calls that the ISR can use:

- *InterruptMask()*
- *InterruptUnmask()*
- *TraceEvent()*

You'll also find these functions (which aren't kernel calls) useful in an ISR:

- *InterruptEnable()* (not recommended)
- *InterruptDisable()* (not recommended)
- *InterruptLock()*

- *InterruptUnlock()*

Let's look at these functions.

To prevent a thread and ISR from interfering with each other, you'll need to tell the kernel to disable interrupts. On a single-processor system, you can simply disable interrupts using the processor's "disable interrupts" opcode. But on an SMP system, disabling interrupts on one processor doesn't disable them on another processor.

The function *InterruptDisable()* (and the reverse, *InterruptEnable()*) performs this operation on a single-processor system. The function *InterruptLock()* (and the reverse, *InterruptUnlock()*) performs this operation on an SMP system.

We recommend that you *always* use the SMP versions of these functions — this makes your code portable to SMP systems, with a negligible amount of overhead.

The *InterruptMask()* and *InterruptUnmask()* functions disable and enable the PIC's recognition of a particular hardware IRQ line. These calls are useful if your interrupt handler ISR is provided by the kernel via *InterruptAttachEvent()* or if you can't clear the cause of the interrupt in a level-sensitive environment quickly. (This would typically be the case if clearing the source of the interrupt is time-consuming — you don't want to spend a lot of time in the interrupt handler. The classic example of this is a floppy-disk controller, where clearing the source of the interrupt may take many milliseconds.) In this case, the ISR would call *InterruptMask()* and schedule a thread to do the actual work. The thread would call *InterruptUnmask()* when it had cleared the source of the interrupt.

Note that these two functions are *counting* — *InterruptUnmask()* must be called the same number of times as *InterruptMask()* in order to have the interrupt source considered enabled again.

The *TraceEvent()* function traces kernel events; you can call it, with some restrictions, in an interrupt handler. For more information, see the System Analysis Toolkit *User's Guide*.

Updating common data structures

Another issue that arises when using interrupts is how to safely update data structures in use between the ISR and the threads in the application. Two important characteristics are worth repeating:

- The ISR runs at a higher priority than any software thread.
- The ISR can't issue kernel calls (except as noted).

This means that you *can't* use thread-level synchronization (such as mutexes, condvars, etc.) in an ISR.

Because the ISR runs at a higher priority than any software thread, it's up to the thread to protect itself against any preemption caused by the ISR. Therefore, the thread should issue *InterruptDisable()* and *InterruptEnable()* calls around any critical data-manipulation operations. Since these calls effectively turn off interrupts, the thread should keep the data-manipulation operations to a bare minimum.

With SMP, there's an additional consideration: one processor could be running the ISR, and another processor could be running a thread related to the ISR. Therefore, on an SMP system, you must use the *InterruptLock()* and *InterruptUnlock()* functions instead. Again, using these functions on a non-SMP system is safe; they'll work just like *InterruptDisable()* and *InterruptEnable()*, albeit with an insignificantly small performance penalty.

Another solution that can be used in some cases to at least guarantee atomic accesses to data elements is to use the *atomic_*()* function calls (below).

Signalling the application code

Since the environment the ISR operates in is very limited, generally you'll want to perform most (if not all) of your actual "servicing" operations at the thread level.

At this point, you have two choices:

- You may decide that some time-critical functionality needs to be done in the ISR, with a thread being scheduled later to do the "real" work.
- You may decide that *nothing* needs to be done in the ISR; you just want to schedule a thread.

This is effectively the difference between *InterruptAttach()* (where an ISR is attached to the IRQ) and *InterruptAttachEvent()* (where a **struct sigevent** is bound to the IRQ).

Let's take a look at the prototype for an ISR function and the *InterruptAttach()* and *InterruptAttachEvent()* functions:

```
int
InterruptAttach (int intr,
                 const struct sigevent * (*handler) (void *, int),
                 const void *area,
                 int size,
                 unsigned flags);

int
InterruptAttachEvent (int intr,
                      const struct sigevent *event,
                      unsigned flags);

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

Using *InterruptAttach()*

Looking at the prototype for *InterruptAttach()*, the function associates the IRQ vector (*intr*) with your ISR handler (*handler*), passing it a communications area (*area*). The *size* and *flags* arguments aren't germane to our discussion here (they're described in the *Library Reference* for the *InterruptAttach()* function).

For the ISR, the *handler()* function takes a **void *** pointer and an **int** identification parameter; it returns a **const struct sigevent *** pointer. The **void *** *area* parameter is the value given to the *InterruptAttach()* function — any value you put in the *area* parameter to *InterruptAttach()* is passed to your *handler()* function. (This is simply a convenient way of coupling the interrupt handler ISR to some data structure. You're certainly free to pass in a **NULL** value if you wish.)

After it has read some registers from the hardware or done whatever processing is required for servicing, the ISR may or may not decide to schedule a thread to actually do the work. In order to schedule a thread, the ISR simply returns a pointer to a **const struct sigevent** structure — the kernel looks at the structure and delivers the event to the destination. (See the *Library Reference* under **sigevent** for a discussion of event types that can be returned.) If the ISR decides not to schedule a thread, it simply returns a **NULL** value.

As mentioned in the documentation for **sigevent**, the event returned can be a signal or a pulse. You may find that a signal or a pulse is satisfactory, especially if you already have a signal or pulse handler for some other reason.

Note, however, that for ISRs we can also return a SIGEV_INTR. This is a special event that really has meaning only for an ISR and its associated *controlling thread*.

A very simple, elegant, and fast way of servicing interrupts from the thread level is to have a thread dedicated to interrupt processing. The thread attaches the interrupt (via *InterruptAttach()*) and then the thread blocks, waiting for the ISR to tell it to do something. Blocking is achieved via the *InterruptWait()* call. This call blocks until the ISR returns a SIGEV_INTR event:

```
main ()
{
    // perform initializations, etc.
    ...
    // start up a thread that is dedicated to interrupt processing
    pthread_create (NULL, NULL, int_thread, NULL);
    ...
    // perform other processing, as appropriate
    ...
}

// this thread is dedicated to handling and managing interrupts
void *
int_thread (void *arg)
{
    // enable I/O privilege
    ThreadCtl (_NTO_TCTL_IO, NULL);
    ...
    // initialize the hardware, etc.
    ...
    // attach the ISR to IRQ 3
    InterruptAttach (IRQ3, isr_handler, NULL, 0, 0);
    ...
    // perhaps boost this thread's priority here
    ...
    // now service the hardware when the ISR says to
    while (1)
    {
        InterruptWait (NULL, NULL);
```

```
        // at this point, when InterruptWait unblocks,
        // the ISR has returned a SIGEV_INTR, indicating
        // that some form of work needs to be done.


        ...
        // do the work
        ...
        // if the isr_handler did an InterruptMask, then
        // this thread should do an InterruptUnmask to
        // allow interrupts from the hardware
    }
}

// this is the ISR
const struct sigevent *
isr_handler (void *arg, int id)
{
    // look at the hardware to see if it caused the interrupt
    // if not, simply return (NULL);
    ...
    // in a level-sensitive environment, clear the cause of
    // the interrupt, or at least issue InterruptMask to
    // disable the PIC from reinterrupting the kernel
    ...
    // return a pointer to an event structure (preinitialized
    // by main) that contains SIGEV_INTR as its notification type.
    // This causes the InterruptWait in "int_thread" to unblock.
    return (&event);
}
```

In the above code sample, we see a typical way of handling interrupts. The main thread creates a special interrupt-handling thread (*int_thread()*). The sole job of that thread is to service the interrupts at the thread level. The interrupt-handling thread attaches an ISR to the interrupt (*isr_handler()*), and then waits for the ISR to tell it to do something. The ISR informs (unblocks) the thread by returning an event structure with the notification type set to SIGEV_INTR.

This approach has a number of advantages over using an event notification type of SIGEV_SIGNAL or SIGEV_PULSE:

- The application doesn't have to have a *MsgReceive()* call (which would be required to wait for a pulse).
- The application doesn't have to have a signal-handler function (which would be required to wait for a signal).
- If the interrupt servicing is critical, the application can create the *int_thread()* thread with a high priority; when the SIGEV_INTR is returned from the *isr_handler()* function, if the *int_thread()* function is of sufficient priority, it runs *immediately*. There's no delay as there might be, for example, between the time that the ISR sent a pulse and another thread eventually called a *MsgReceive()* to get it.

The only caveat to be noted when using *InterruptWait()* is that the thread that *attached* the interrupt is the one that must *wait* for the SIGEV_INTR.

Using *InterruptAttachEvent()*

Most of the discussion above for *InterruptAttach()* applies to the *InterruptAttachEvent()* function, with the obvious exception of the ISR. You don't provide an ISR in this case — the kernel notes that you called *InterruptAttachEvent()* and handles the interrupt itself. Since you also bound a **`struct sigevent`** to the IRQ, the kernel can now dispatch the event. The major advantage is that we avoid a context switch into the ISR and back.

An important point to note is that the kernel automatically performs an *InterruptMask()* in the interrupt handler. Therefore, it's up to you to perform an *InterruptUnmask()* when you actually clear the source of the interrupt in your interrupt-handling thread. This is why *InterruptMask()* and *InterruptUnmask()* are counting.

Running out of interrupt events

If you're working with interrupts, you might see an **`Out of Interrupt Events`** error. This happens when the system is no longer able to run user code and is stuck in the kernel, most frequently because:

- The interrupt load is too high for the CPU (it's spending all of the time handling the interrupt).

  Or:
- There's an interrupt handler — one connected with *InterruptAttach()*, not *InterruptAttachEvent()* — that doesn't properly clear the interrupt condition from the device (leading to the case above).

If you call *InterruptAttach()* in your code, look at the handler code first and make sure you're properly clearing the interrupt condition from the device before returning to the OS.

If you encounter this problem, even with all hardware interrupts disabled, it could be caused by misuse or excessive use of software timers.

Problems with shared interrupts

It's possible for different devices to share an interrupt (for example if you've run out of hardware interrupt lines), but we don't recommend you do this with hardware that will be generating a lot of interrupts. We also recommend you not share interrupts with drivers that you don't have complete source control over, because you need to be sure that the drivers process interrupts properly.

Sharing interrupts can decrease your performance, because when the interrupt fires, *all* of the devices sharing the interrupt need to run and check to see if it's for them. Many drivers read the registers in their interrupt handlers to see if the interrupt is really for them, and then ignore it if it isn't. But some drivers don't; they schedule their thread-level event handlers to check their hardware, which is inefficient and reduces performance.

If you have a frequent interrupt source sharing an interrupt with a driver that schedules a thread to check the hardware, the overhead of scheduling the thread becomes noticeable.

Sharing interrupts can increase interrupt latency, depending upon exactly what each of the drivers does. After an interrupt fires, the kernel doesn't reenable it until *all* driver handlers tell the kernel that they've finished handling it. If one driver takes a long time servicing a shared interrupt that's masked, and another device on the same interrupt causes an interrupt during that time period, the processing of that interrupt can be delayed for an unknown length of time.

Advanced topics

Now that we've seen the basics of handling interrupts, let's take a look at some more details and some advanced topics.

Interrupt environment

When your ISR is running, it runs in the context of the process that attached it, except with a different stack. Since the kernel uses an internal interrupt-handling stack for hardware interrupts, your ISR is impacted in that the internal stack is small. Generally, you can assume that you have about 200 bytes available.

The PIC doesn't get the EOI command until *after* all ISRs — whether supplied by your code via *InterruptAttach()* or by the kernel if you use *InterruptAttachEvent()* — for that particular interrupt have been run. Then the kernel itself issues the EOI; your code should *not* issue the EOI command.

Ordering of shared interrupts

If you're using interrupt sharing, then by default when you attach an ISR using *InterruptAttach()* or *InterruptAttachEvent()*, the new ISR goes to the beginning of the list of ISRs for that interrupt. You can specifically request that your ISR be placed at the end of the list by specifying a *flags* argument of _NTO_INTR_FLAGS_END.

Note that there's no way to specify any other order (e.g. middle, fifth, second, etc.).

Interrupt latency

Another factor of concern for realtime systems is the amount of time taken between the generation of the hardware interrupt and the first line of code executed by the ISR. There are two factors to consider here:

- If any thread in the system calls *InterruptDisable()* or *InterruptLock()*, then no interrupts are processed until the *InterruptEnable()* or *InterruptUnlock()* function call is issued.

- In any event, if interrupts are enabled, the kernel begins executing the first line of the *first* ISR (in case multiple ISRs are associated with an interrupt) in short order (e.g. under 21 CPU instructions on an x86).

Atomic operations

Some convenience functions are defined in the include file **`<atomic.h>`** — these allow you to perform atomic operations (i.e. operations that are guaranteed to be indivisible or uninterruptible).

Using these functions alleviates the need to disable and enable interrupts around certain small, well-defined operations with variables, such as:

- adding a value
- subtracting a value
- clearing bits
- setting bits
- toggling bits

Variables used in an ISR must be marked as "**`volatile`**".

See the *Library Reference* under *atomic_*()* for more information.

Chapter 7

Heap Analysis: Making Memory Errors a Thing of the Past

In this chapter. . .

Introduction

If you develop a program that dynamically allocates memory, you're also responsible for tracking any memory that you allocate whenever a task is performed, and for releasing that memory when it's no longer required. If you fail to track the memory correctly, you may introduce "memory leaks" or unintentionally write to an area outside of the memory space.

Conventional debugging techniques usually prove to be ineffective for locating the source of corruption or leak because memory-related errors typically manifest themselves in an unrelated part of the program. Tracking down an error in a multithreaded environment becomes even more complicated because the threads all share the same memory address space.

In this chapter, we'll describe how Neutrino manages the heap and introduce you to a special version of our memory management functions that will help you to diagnose your memory management problems.

Dynamic memory management

In a program, you'll dynamically request memory buffers or blocks of a particular size from the runtime environment using *malloc()*, *realloc()*, or *calloc()*, and then you'll release them back to the runtime environment when they're no longer required using *free()*.

The *memory allocator* ensures that your requests are satisfied by managing a region of the program's memory area known as the *heap*. In this heap, it tracks all of the information — such as the size of the original block — about the blocks and heap buffers that it has allocated to your program, in order that it can make the memory available to you during subsequent allocation requests. When a block is released, it places it on a list of available blocks called a *free list*. It usually keeps the information about a block in the header that precedes the block itself in memory.

The runtime environment grows the size of the heap when it no longer has enough memory available to satisfy allocation requests, and it returns memory from the heap to the system when the program releases memory.

The basic heap allocation mechanism is broken up into two separate pieces, a chunk-based small block allocator and a list-based large block allocator. By configuring specific parameters, you can select the various sizes for the chunks in the chunk allocator and also the boundary between the small and the large allocator.

Arena allocations

Both the small and the large portions of the allocator allocate and deallocate memory from the system in the form of *arena* chunks by calling *mmap()* and *munmap()*. By default, the arena allocations are performed in 32 KB chunks. This value is specified by one of the following:

- a global variable that's defined in the allocator, but can be redefined or modified in the application
- the *_amblksiz* global variable

This value must be a multiple of 4 KB, and currently is limited to being less than 256 KB. You can also configure this parameter by setting the **MALLOC_ARENA_SIZE** environment variable, or by calling *mallopt()* with MALLOC_ARENA_SIZE as the command.

For example, if you want to change the arena size to 16 KB, do one of the following:

- **`_amblksiz = 16384;`**
- **`export MALLOC_ARENA_SIZE=16384`**
- **`mallopt(MALLOC_ARENA_SIZE, 16384);`**

Environment variables are checked only at program startup, but changing them is the easiest way of configuring parameters for the allocator so that these parameters are used for allocations that occur before *main()*.

The allocator also attempts to cache recently used arena blocks. This cache is shared between the small- and the large-block allocator. You can configure the arena cache by setting the following environment variables:

MALLOC_ARENA_CACHE_MAXBLK

The number of cached arena blocks.

MALLOC_ARENA_CACHE_MAXSZ

The total size of the cached arena blocks.

Alternatively, you can call:

```
mallopt(MALLOC_ARENA_CACHE_MAXSZ, size);
mallopt(MALLOC_ARENA_CACHE_MAXBLK, number);
```

You can tell the allocator to never release memory back to the system from its arena cache by setting the environment variable:

```
export MALLOC_MEMORY_HOLD=1
```

or by calling:

```
mallopt(MALLOC_MEMORY_HOLD, 1);
```

Once you've changed the values by using *mallopt()* for either MALLOC_ARENA_CACHE_MAXSZ or MALLOC_ARENA_CACHE_MAXBLK, you must call *mallopt()* to cause the arena cache to be adjusted immediately:

```
// Adjust the cache to the current parameters or
// release all cache blocks, but don't change parameters
mallopt(MALLOC_ARENA_CACHE_FREE_NOW, 0);
mallopt(MALLOC_ARENA_CACHE_FREE_NOW, 1);
```

Without a call with a command of MALLOC_ARENA_CACHE_FREE_NOW, the changes made to the cache parameters will take effect whenever memory is subsequently released to the cache.

So for example, if the arena cache currently has 10 blocks, for a total size of say 320 KB, and if you change the arena parameters to MALLOC_ARENA_CACHE_MAXBLK = 5 and MALLOC_ARENA_CACHE_MAXSZ = 200 KB, an immediate call to **`mallopt(MALLOC_ARENA_CACHE_FREE_NOW, 0)`** will reduce the cache so that the number of blocks is no more than 5, and the total cache size is no more than 320 KB. If you don't make the call to *mallopt()*, then no immediate changes are made. If the application frees some memory, causing a new arena of size 32 KB to get released to the system, this will not be cached, but will be released to the system immediately.

You can use **MALLOC_ARENA_CACHE_MAXSZ** and **MALLOC_ARENA_CACHE_MAXBLK** either together or independently. A value of zero is ignored.

You can preallocate and populate the arena cache by setting the **MALLOC_MEMORY_PREALLOCATE** environment variable to a value that specifies the size of the total arena cache. The cache is populated by multiple arena allocation calls in chunks whose size is specified by the value of **MALLOC_ARENA_SIZE**.

The preallocation option doesn't alter the **MALLOC_ARENA_CACHE_MAXBLK** and **MALLOC_ARENA_CACHE_MAXSZ** options. So if you preallocate 10 MB of memory in cache blocks, to ensure that this memory stays in the application throughout the lifetime of the application, you should also set the values of **MALLOC_ARENA_CACHE_MAXBLK** and **MALLOC_ARENA_CACHE_MAXSZ** to something appropriate.

Small block configuration

You configure the small blocks by setting various bands of different sizes. Each band defines a fixed size block, and a number that describes the size of the pool for that size. The allocator initially adjusts all band sizes to be multiples of _MALLOC_ALIGN (which is 8), and then takes the size of the pools and normalizes them so that each band pool is constructed from a pool size of 4 KB.

By default, bands in the allocator are defined as:

- _MALLOC_ALIGN × 2 = 16
- _MALLOC_ALIGN × 3 = 24
- _MALLOC_ALIGN × 4 = 32
- _MALLOC_ALIGN × 6 = 48
- _MALLOC_ALIGN × 8 = 64
- _MALLOC_ALIGN × 10 = 80
- _MALLOC_ALIGN × 12 = 96

- _MALLOC_ALIGN × 16 = 128

so the smallest small block is 16 bytes, and the largest small block is 128 bytes. Allocations larger than the largest band size are serviced by the large allocator.

After initial normalization by the allocator, the band sizes and the pool sizes are adjusted to the following:

Band size	Number of items
16	167
24	125
32	100
48	71
64	55
80	45
96	38
128	28

This normalization takes into account alignment restrictions and overhead needed by the allocator to manage these blocks. The number of items is the number of blocks of the given size that are created each time a new "bucket" is allocated.

You can specify you own band configurations by defining the following in your application's code:

```
typedef struct Block Block;
typedef struct Band Band;

struct Band {
  short nbpe; /* element size */
  short nalloc; /* elements per block */
  size_t slurp;
  size_t esize;
  size_t mem;
  size_t rem;
  unsigned nalloc_stats;
  Block * alist;  /* Blocks that have data to allocate */
  Block * dlist;  /* completely allocated (depleted) Blocks */
  unsigned    blk_alloced;    /* #blocks allocated */
  unsigned    blk_freed;      /* #blocks freed */
  unsigned    alloc_counter;  /* allocs */
  unsigned    free_counter;   /* frees */
  unsigned    blk_size;     /* size of allocated blocks */
};

static Band a1 = { _MALLOC_ALIGN*2, 32, 60};
static Band a2 = { _MALLOC_ALIGN*3, 32, 60};
static Band a3 = { _MALLOC_ALIGN*4, 32, 60};
static Band a4 = { _MALLOC_ALIGN*5, 24, 60};
static Band a5 = { _MALLOC_ALIGN*6, 24, 60};
static Band a6 = { _MALLOC_ALIGN*7, 24, 60};
```

```
static Band a7 = { _MALLOC_ALIGN*8, 16, 60};
static Band a8 = { _MALLOC_ALIGN*9, 8, 60};
static Band a9 = { _MALLOC_ALIGN*10, 8, 60};
static Band a10 = { _MALLOC_ALIGN*11, 8, 60};
static Band a11 = { _MALLOC_ALIGN*12, 8, 60};
static Band a12 = { _MALLOC_ALIGN*13, 8, 60};
static Band a13 = { _MALLOC_ALIGN*32, 10, 60};
Band *__dynamic_Bands[] = { &a1, &a2, &a3, &a4, &a5, &a6,
                            &a7, &a8, &a9, &a10, &a11, &a12,
                            &a13,
                          };
unsigned __dynamic_nband=13;
```

The main variables are *__dynamic_Bands[]* and *__dynamic_Bands*, which specify the band configurations and the number of bands. For example, the following line:

```
static Band a9 = { _MALLOC_ALIGN*10, 8, 60};
```

specifies a band size of 80 bytes, with each chunk having at least 8 blocks, and a preallocation value of 60. The allocator first normalizes the band size to 80, and the number of items to 45. Then during initialization of the allocator, it preallocates at least 60 blocks of this size band. (Each bucket will have 45 blocks, so 60 blocks will be constructed from two buckets).

If you specify your own bands:

- The sizes must all be distinct.
- The band configuration must be provided in ascending order of sizes (i.e band 0 size **<** band 1 size **<** band 2 size, and so on).

A user-specified band configuration of:

```
static Band a1 = { 2, 32, 60};
static Band a2 = { 15, 32, 60};
static Band a3 = { 29, 32, 60};
static Band a4 = { 55, 24, 60};
static Band a5 = { 100, 24, 60};
static Band a6 = { 130, 24, 60};
static Band a7 = { 260, 8, 60};
static Band a8 = { 600, 4, 60};
Band *__dynamic_Bands[] = {&a1, &a2, &a3, &a4,
                           &a5, &a6, &a7, &a8,
                          };
unsigned __dynamic_nband=8;
```

will be normalized to:

Band size	Number of items
8	251
16	167

continued…

Band size	Number of items
32	100
56	62
104	35
136	27
264	13
600	5

In addition to specifying the band configurations, you also have to set the environment variable:

```
export MALLOC_MEMORY_BANDCONFIG=1
```

to ensure that your configurations are picked.

For the above configuration, allocations larger than 600 bytes will be serviced by the large block allocator.

When used in conjunction with the **MALLOC_MEMORY_PREALLOCATE** option for the arena cache, the preallocations of blocks in bands are performed by initially populating the arena cache, and then allocating bands from this arena cache.

You can also configure the bands by using the **MALLOC_BAND_CONFIG_STR** environment variable. The string format is:

```
N:s1,n1,p1:s2,n2,p2:s3,n3,p3: ... :sN,nN,pN
```

where the components are:

s The band size.

n The number of items.

p The preallocation value, which can be zero.

You must specify *s*, *n*, and *p* for each band. The string can't include any spaces; the only valid characters are digits, colons (`:`), and commas (`,`). Position is important. The parsing is simple and strict: sizes are assumed to be provided in ascending order, further validation is done by the allocator. If the allocator doesn't like the string, it ignores it completely.

Heap corruption

Heap corruption occurs when a program damages the allocator's view of the heap. The outcome can be relatively benign and cause a memory leak (where some memory isn't returned to the heap and is inaccessible to the program afterward), or it may be fatal and cause a memory fault, usually within the allocator itself. A memory fault typically occurs within the allocator when it manipulates one or more of its free lists after the heap has been corrupted.

It's especially difficult to identify the source of corruption when the source of the fault is located in another part of the code base. This is likely to happen if the fault occurs when:

- a program attempts to free memory
- a program attempts to allocate memory after it's been freed
- the heap is corrupted long before the release of a block of memory
- the fault occurs on a subsequent block of memory
- contiguous memory blocks are used
- your program is multithreaded
- the memory allocation strategy changes

Contiguous memory blocks

When contiguous blocks are used, a program that writes outside of the bounds can corrupt the allocator's information about the block of memory it's using, as well as the allocator's view of the heap. The view may include a block of memory that's before or after the block being used, and it may or may not be allocated. In this case, a fault in the allocator will likely occur during an unrelated attempt to allocate or release memory.

Multithreaded programs

Multithreaded execution may cause a fault to occur in a different thread from the thread that actually corrupted the heap, because threads interleave requests to allocate or release memory.

When the source of corruption is located in another part of the code base, conventional debugging techniques usually prove to be ineffective. Conventional debugging typically applies breakpoints — such as stopping the program from executing — to narrow down the offending section of code. While this may be effective for single-threaded programs, it's often unyielding for multithreaded execution because the fault may occur at an unpredictable time, and the act of debugging the program may influence the appearance of the fault by altering the way that thread execution occurs. Even when the source of the error has been narrowed down, there may be a substantial amount of manipulation performed on the block before it's released, particularly for long-lived heap buffers.

Allocation strategy

A program that works in a particular memory allocation strategy may abort when the allocation strategy is changed in a minor way. A good example of this is a memory overrun condition (for more information see "Overrun and underrun errors," below) where the allocator is permitted to return blocks that are larger than requested in order to satisfy allocation requests. Under this circumstance, the program may behave normally in the presence of overrun conditions. But a simple change, such as changing the size of the block requested, may result in the allocation of a block of the exact size requested, resulting in a fatal error for the offending program.

Fatal errors may also occur if the allocator is configured slightly differently, or if the allocator policy is changed in a subsequent release of the runtime library. This makes it all the more important to detect errors early in the life cycle of an application, even if it doesn't exhibit fatal errors in the testing phase.

Common sources

Some of the most common sources of heap corruption include:

- a memory assignment that corrupts the header of an allocated block
- an incorrect argument that's passed to a memory allocation function
- an allocator that made certain assumptions in order to avoid keeping additional memory to validate information, or to avoid costly runtime checking
- invalid information that's passed in a request, such as to *free()*
- overrun and underrun errors
- releasing memory
- using uninitialized or stale pointers

Even the most robust allocator can occasionally fall prey to the above problems. Let's take a look at the last three items in more detail.

Overrun and underrun errors

Overrun and underrun errors occur when your program writes outside of the bounds of the allocated block. They're one of the most difficult type of heap corruption to track down, and usually the most fatal to program execution.

Overrun errors occur when the program writes *past* the end of the allocated block. Frequently this causes corruption in the next contiguous block in the heap, whether or not it's allocated. When this occurs, the behavior that's observed varies depending on whether that block is allocated or free, and whether it's associated with a part of the program related to the source of the error. When a neighboring block that's allocated becomes corrupted, the corruption is usually apparent when that block is released elsewhere in the program. When an unallocated block becomes corrupted, a fatal error will usually result during a subsequent allocation request. Although this may well be the next allocation request, it actually depends on a complex set of conditions that could result in a fault at a much later point in time, in a completely unrelated section of the program, especially when small blocks of memory are involved.

Underrun errors occur when the program writes *before* the start of the allocated block. Often they corrupt the header of the block itself, and sometimes, the preceding block in memory. Underrun errors usually result in a fault that occurs when the program attempts to release a corrupted block.

Releasing memory

In order to release memory, your program must track the pointer for the allocated block and pass it to the *free()* function. If the pointer is stale, or if it doesn't point to the exact start of the allocated block, it may result in heap corruption.

A pointer is *stale* when it refers to a block of memory that's already been released. A duplicate request to *free()* involves passing *free()* a stale pointer — there's no way to know whether this pointer refers to unallocated memory, or to memory that's been used to satisfy an allocation request in another part of the program.

Passing a stale pointer to *free()* may result in a fault in the allocator, or worse, it may release a block that's been used to satisfy another allocation request. If this happens, the code making the allocation request may compete with another section of code that subsequently allocated the same region of heap, resulting in corrupted data for one or both. The most effective way to avoid this error is to NULL out pointers when the block is released, but this is uncommon, and difficult to do when pointers are aliased in any way.

A second common source of errors is to attempt to release an interior pointer (i.e. one that's somewhere inside the allocated block rather than at the beginning). This isn't a legal operation, but it may occur when the pointer has been used in conjunction with pointer arithmetic. The result of providing an interior pointer is highly dependent on the allocator and is largely unpredictable, but it frequently results in a fault in the *free()* call.

A more rare source of errors is to pass an uninitialized pointer to *free()*. If the uninitialized pointer is an automatic (stack) variable, it may point to a heap buffer, causing the types of coherency problems described for duplicate *free()* requests above. If the pointer contains some other non-NULL value, it may cause a fault in the allocator.

Using uninitialized or stale pointers

If you use uninitialized or stale pointers, you might corrupt the data in a heap buffer that's allocated to another part of the program, or see memory overrun or underrun errors.

Detecting and reporting errors

The primary goal for detecting heap corruption problems is to correctly identify the source of the error, to avoid getting a fault in the allocator at some later point in time.

A first step to achieving this goal is to create an allocator that's able to determine whether the heap was corrupted on every entry into the allocator, whether it's for an allocation request or for a release request. For example, on a release request, the allocator should be capable of determining whether:

- the pointer given to it is valid
- the associated block's header is corrupt
- either of the neighboring blocks is corrupt

To achieve this goal, we'll use a replacement library for the allocator that can keep additional block information in the header of every heap buffer. You can use this library while testing the application to help isolate any heap corruption problems. When this allocator detects a source of heap corruption, it can print an error message indicating:

- the point at which the error was detected
- the program location that made the request
- information about the heap buffer that contained the problem

The library technique can be refined to also detect some of the sources of errors that may still elude detection, such as memory overrun or underrun errors, that occur before the corruption is detected by the allocator. This may be done when the standard libraries are the vehicle for the heap corruption, such as an errant call to *memcpy()*, for example. In this case, the standard memory manipulation functions and string functions can be replaced with versions that make use of the information in the debugging allocator library to determine if their arguments reside in the heap, and whether they would cause the bounds of the heap buffer to be exceeded. Under these conditions, the function can then call the error-reporting functions to provide information about the source of the error.

Using the `malloc` debug library

The **`malloc`** debug library provides the capabilities described in the above section. It's available when you link to either the normal memory allocator library, or to the debug library:

To access:	Link using this option:
Nondebug library	**`-lmalloc`**
Debug library	**`-lmalloc_g`**

If you use the debug library, you must also include **`/usr/lib/malloc_g`** as the first entry of your **LD_LIBRARY_PATH** environment variable before running your application.

> This library was updated in QNX Momentics 6.3.0 SP2.

Another way to use the debug **`malloc`** library is to use the **LD_PRELOAD** capability to the dynamic loader. The **LD_PRELOAD** environment variable lets you specify libraries to load prior to any other library in the system. In this case, set the **LD_PRELOAD** variable to point to the location of the debug **malloc** library (or the nondebug one as the case may be), by saying:

```
LD_PRELOAD=/usr/lib/malloc_g/libmalloc.so.2
```

or:

```
LD_PRELOAD=/usr/lib/libmalloc.so.2
```

In this chapter, all references to the **malloc** library refer to the debug version, unless otherwise specified.

Both versions of the library share the same internal shared object name, so it's actually possible to link against the nondebug library and test using the debug library when you run your application. To do this, you must change the **LD_LIBRARY_PATH** as indicated above.

The nondebug library doesn't perform heap checking; it provides the same memory allocator as the system library.

By default, the **malloc** library provides a minimal level of checking. When an allocation or release request is performed, the library checks only the immediate block under consideration and its neighbors, looking for sources of heap corruption.

Additional checking and more informative error reporting can be done by using additional calls provided by the **malloc** library. The *mallopt()* function provides control over the types of checking performed by the library. There are also debug versions of each of the allocation and release routines that you can use to provide both file and line information during error-reporting. In addition to reporting the file and line information about the caller when an error is detected, the error-reporting mechanism prints out the file and line information that was associated with the allocation of the offending heap buffer.

To control the use of the **malloc** library and obtain the correct prototypes for all the entry points into it, it's necessary to include a different header file for the library. This header file is included in **<malloc_g/malloc.h>**. If you want to use any of the functions defined in this header file, other than *mallopt()*, make sure that you link your application with the debug library. If you forget, you'll get undefined references during the link.

The recommended practice for using the library is to always use the library for debug variants in builds. In this case, the macro used to identify the debug variant in C code should trigger the inclusion of the **<malloc_g/malloc.h>** header file, and the **malloc** debug library option should always be added to the link command. In addition, you may want to follow the practice of always adding an exit handler that provides a dump of leaked memory, and initialization code that turns on a reasonable level of checking for the debug variant of the program.

The **malloc** library achieves what it needs to do by keeping additional information in the header of each heap buffer. The header information includes additional storage for keeping doubly-linked lists of all allocated blocks, file, line, and other debug information, flags and a CRC of the header. The allocation policies and configuration are identical to the normal system memory allocation routines except for the additional internal overhead imposed by the **`malloc`** library. This allows the **`malloc`** library to perform checks without altering the size of blocks requested by the program. Such manipulation could result in an alteration of the behavior of the program with respect to the allocator, yielding different results when linked against the **`malloc`** library.

All allocated blocks are integrated into a number of allocation chains associated with allocated regions of memory kept by the allocator in *arenas* or *blocks*. The **`malloc`** library has intimate knowledge about the internal structures of the allocator, allowing it to use short cuts to find the correct heap buffer associated with any pointer, resorting to a lookup on the appropriate allocation chain only when necessary. This minimizes the performance penalty associated with validating pointers, but it's still significant.

The time and space overheads imposed by the **`malloc`** library are too great to make it suitable for use as a production library, but are manageable enough to allow them to be used during the test phase of development and during program maintenance.

What's checked?

As indicated above, the **`malloc`** library provides a minimal level of checking by default. This includes a check of the integrity of the allocation chain at the point of the local heap buffer on every allocation request. In addition, the flags and CRC of the header are checked for integrity. When the library can locate the neighboring heap buffers, it also checks their integrity. There are also checks specific to each type of allocation request that are done. Call-specific checks are described according to the type of call below.

You can enable additional checks by using the *mallopt()* call. For more information on the types of checking, and the sources of heap corruption that can be detected, see of "Controlling the level of checking," below.

Allocating memory

When a heap buffer is allocated using any of the heap-allocation routines, the heap buffer is added to the allocation chain for the *arena* or *block* within the heap that the heap buffer was allocated from. At this time, any problems detected in the allocation chain for the arena or block are reported. After successfully inserting the allocated buffer in the allocation chain, the previous and next buffers in the chain are also checked for consistency.

Reallocating memory

When an attempt is made to resize a buffer through a call to the *realloc()* function, the pointer is checked for validity if it's a non-NULL value. If it's valid, the header of the heap buffer is checked for consistency. If the buffer is large enough to satisfy the request, the buffer header is modified, and the call returns. If a new buffer is required to satisfy the request, memory allocation is performed to obtain a new buffer large enough to satisfy the request with the same consistency checks being applied as in the case of memory allocation described above. The original buffer is then released.

If fill-area boundary checking is enabled (described in the "Manual Checking" section) the guard code checks are also performed on the allocated buffer before it's actually resized. If a new buffer is used, the guard code checks are done just before releasing the old buffer.

Releasing memory

This includes, but isn't limited to, checking to ensure that the pointer provided to a *free()* request is correct and points to an allocated heap buffer. Guard code checks may also be performed on release operations to allow fill-area boundary checking.

Controlling the level of checking

The *mallopt()* function call allows extra checks to be enabled within the library. The call to *mallopt()* requires that the application be aware that the additional checks are programmatically enabled. The other way to enable the various levels of checking is to use environment variables for each of the *mallopt()* options. Using environment variables lets you specify options that will be enabled from the time the program runs, as opposed to only when the code that triggers these options to be enabled (i.e. the *mallopt()* call) is reached. For certain programs that perform a lot of allocations before *main()*, setting options using *mallopt()* calls from *main()* or after that may be too late. In such cases, it's better to use environment variables.

The prototype of *mallopt()* is:

```
int mallopt ( int cmd,
              int value );
```

The arguments are:

cmd
: The command you want to use. The options used to enable additional checks in the library are:
  - MALLOC_CKACCESS
  - MALLOC_FILLAREA
  - MALLOC_CKCHAIN

  We look at some of the other commands later in this chapter.

value
: A value corresponding to the command used.

For more details, see the entry for *mallopt()* in the Neutrino *Library Reference*.

Description of optional checks

MALLOC_CKACCESS

Turn on (or off) boundary checking for memory and string operations. Environment variable: **MALLOC_CKACCESS**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

This helps to detect buffer overruns and underruns that are a result of memory or string operations. When on, each pointer operand to a memory or string operation is checked to see if it's a heap buffer. If it is, the size of the heap buffer is checked, and the information is used to ensure that no assignments are made beyond the bounds of the heap buffer. If an attempt is made that would assign past the buffer boundary, a diagnostic warning message is printed.

Here's how you can use this option to find an overrun error:

```
...
char *p;
int opt;
opt = 1;
mallopt(MALLOC_CKACCESS, opt);
p = malloc(strlen("hello"));
strcpy(p, "hello, there!");  /* a warning is generated
                                         here */
...
```

The following illustrates how access checking can trap a reference through a stale pointer:

```
...

char *p;
int opt;
opt = 1;
mallopt(MALLOC_CKACCESS, opt);
p = malloc(30);
free(p);
strcpy(p, "hello, there!");
```

MALLOC_FILLAREA

Turn on (or off) fill-area boundary checking that validates that the program hasn't overrun the user-requested size of a heap buffer. Environment variable: **MALLOC_FILLAREA**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

It does this by applying a guard code check when the buffer is released or when it's resized. The guard code check works by filling any excess space available at the end of the heap buffer with a pattern of bytes. When the buffer is released or resized, the trailing portion is checked to see if the pattern is still present. If not, a diagnostic warning message is printed.

The effect of turning on fill-area boundary checking is a little different than enabling other checks. The checking is performed only on memory buffers allocated after the point in time at which the check was enabled. Memory buffers allocated before the change won't have the checking performed.

Here's how you can catch an overrun with the fill-area boundary checking option:

```
...
...
int *foo, *p, i, opt;
opt = 1;
mallopt(MALLOC_FILLAREA, opt);
foo = (int *)malloc(10*4);
for (p = foo, i = 12; i > 0; p++, i--)
    *p = 89;
free(foo);  /* a warning is generated here */
```

MALLOC_CKCHAIN

Enable (or disable) full chain checking. This option is expensive and should be considered as a last resort when some code is badly corrupting the heap and otherwise escapes the detection of boundary checking or fill-area boundary checking. Environment variable: **MALLOC_CKCHAIN**.

The *value* argument can be:

- zero to disable the checking
- nonzero to enable it

This kind of corruption can occur under a number of circumstances, particularly when they're related to direct pointer assignments. In this case, the fault may occur before a check such as fill-area boundary checking can be applied. There are also circumstances in which both fill-area boundary checking and the normal attempts to check the headers of neighboring buffer fail to detect the source of the problem. This may happen if the buffer that's overrun is the first or last buffer associated with a block or arena. It may also happen when the allocator chooses to satisfy some requests, particularly those for large buffers, with a buffer that exactly fits the program's requested size.

Full-chain checking traverses the entire set of allocation chains for all arenas and blocks in the heap every time a memory operation (including allocation requests) is performed. This lets the developer narrow down the search for a source of corruption to the nearest memory operation.

Forcing verification

You can force a full allocation chain check at certain points while your program is executing, without turning on chain checking. Specify the following option for *cmd*:

MALLOC_VERIFY

Perform a chain check immediately. If an error is found, perform error handling. The *value* argument is ignored.

Specifying an error handler

Typically, when the library detects an error, a diagnostic message is printed and the program continues executing. In cases where the allocation chains or another crucial part of the allocator's view is hopelessly corrupted, an error message is printed and the program is aborted (via *abort()*).

You can override this default behavior by specifying what to do when a warning or a fatal condition is detected:

cmd The error handler to set; one of:

MALLOC_FATAL Specify the malloc fatal handler. Environment variable: **MALLOC_FATAL**.

MALLOC_WARN Specify the malloc warning handler handler. Environment variable: **MALLOC_WARN**.

value An integer value that indicates which one of the standard handlers provided by the library to use:

M_HANDLE_ABORT

Terminate execution with a call to *abort()*.

M_HANDLE_EXIT Exit immediately.

M_HANDLE_IGNORE

Ignore the error and continue.

M_HANDLE_CORE Cause the program to dump a core file.

M_HANDLE_SIGNAL

Stop the program when this error occurs, by sending it a stop signal (SIGSTOP). This lets you attach to this process using a debugger. The program is stopped inside the error-handler function, and a backtrace from there should show you the exact location of the error.

If you use environment variables to specify options to the `malloc` library for either **MALLOC_FATAL** or **MALLOC_WARN**, you must pass the value that indicates the handler, not its symbolic name:

Handler	Value
M_HANDLE_IGNORE	0
M_HANDLE_ABORT	1

continued. . .

Handler	Value
M_HANDLE_EXIT	2
M_HANDLE_CORE	3
M_HANDLE_SIGNAL	4

These values are also defined in **/usr/include/malloc_g/malloc-lib.h**.

M_HANDLE_CORE and M_HANDLE_SIGNAL were added in QNX Momentics 6.3.0 SP2.

You can OR any of these handlers with the value, **MALLOC_DUMP**, to cause a complete dump of the heap before the handler takes action.

Here's how you can cause a memory overrun error to abort your program:

```
...
int *foo, *p, i;
int opt;
opt = 1;
mallopt(MALLOC_FILLAREA,  opt);
foo = (int *)malloc(10*4);
for (p = foo, i = 12; i > 0; p++, i--)
    *p = 89;
opt = M_HANDLE_ABORT;
mallopt(MALLOC_WARN, opt);
free(foo); /* a fatal error is generated here */
```

Other environment variables

MALLOC_INITVERBOSE

Enable some initial verbose output regarding other variables that are enabled.

MALLOC_BTDEPTH

Set the depth of the backtrace for allocations (i.e. where the allocation occurred) on CPUs that support deeper backtrace levels. Currently the builtin-return-address feature of **gcc** is used to implement deeper backtraces for the debug **malloc** library. The default value is 0.

MALLOC_TRACEBT

Set the depth of the backtrace for errors and warnings on CPUs that support deeper backtrace levels. Currently the builtin-return-address feature of **gcc** is used to implement deeper backtraces for the debug **malloc** library. The default value is 0.

MALLOC_DUMP_LEAKS

Trigger leak detection on exit of the program. The output of the leak detection is sent to the file named by this variable.

MALLOC_TRACE

Enable tracing of all calls to *malloc()*, *free()*, *calloc()*, *realloc()*, etc. A trace of the various calls is store in the file named by this variable.

MALLOC_CKACCESS_LEVEL

Specify the level of checking performed by the **MALLOC_CKACCESS** option. By default, a basic level of checking is performed. By increasing the level of checking, additional things that could be errors are also flagged. For example, a call to *memset()* with a length of zero is normally safe, since no data is actually moved. If the arguments, however, point to illegal locations (memory references that are invalid), this normally suggests a case where there is a problem potentially lurking inside the code. By increasing the level of checking, these kinds of errors are also flagged.

These environment variables were added in QNX Momentics 6.3.0 SP2.

Caveats

The debug **malloc** library, when enabled with various checking, uses more stack space (i.e. calls more functions, uses more local variables etc.) than the regular **libc** allocator. This implies that programs that explicitly set the stack size to something smaller than the default may encounter problems such as running out of stack space. This may cause the program to crash. You can prevent this by increasing the stack space allocated to the threads in question.

MALLOC_FILLAREA is used to do fill-area checking. If fill-area checking isn't enabled, the program can't detect certain types of errors. For example, if an application accesses beyond the end of a block, and the real block allocated by the allocator is larger than what was requested, the allocator won't flag an error unless **MALLOC_FILLAREA** is enabled. By default, this checking isn't enabled.

MALLOC_CKACCESS is used to validate accesses to the **str*** and **mem*** family of functions. If this variable isn't enabled, such accesses won't be checked, and errors aren't reported. By default, this checking isn't enabled.

MALLOC_CKCHAIN performs extensive heap checking on every allocation. When you enable this environment variable, allocations can be much slower. Also since full heap checking is performed on every allocation, an error anywhere in the heap could be reported upon entry into the allocator for any operation. For example, a call to **free(x)** will check block *x* as well as the complete heap for errors before completing the operation (to free block *x*). So any error in the heap will be reported in the context of freeing block *x*, even if the error itself isn't specifically related to this operation.

When the debug library reports errors, it doesn't always exit immediately; instead it continues to perform the operation that causes the error, and corrupts the heap (since the operation that raises the warning is actually an illegal operation). You can control this behavior by using the MALLOC_WARN and MALLOC_FATAL handler described earlier. If specific handlers are not provided, the heap will be corrupted and other errors could result and be reported later because of the first error. The best solution is to focus on the first error and fix it before moving onto other errors. See the description of **MALLOC_CKCHAIN** for more information on how these errors may end up getting reported.

Although the debug **malloc** library allocates blocks to the process using the same algorithms as the standard allocator, the library itself requires additional storage to maintain block information, as well as to perform sanity checks. This means that the layout of blocks in memory using the debug allocator is slightly different than with the standard allocator.

If you use certain optimization options such as **-O1**, **-O2**, or **-O3**, the debug **malloc** library won't work correctly because these options make **gcc** use builtin versions of some functions, such as *strcpy()* and *strcmp()*. Use the **-fno-builtin** option to prevent this.

Manual checking (bounds checking)

There are times when it may be desirable to obtain information about a particular heap buffer or print a diagnostic or warning message related to that heap buffer. This is particularly true when the program has its own routines providing memory manipulation and you wish to provide bounds checking. This can also be useful for adding additional bounds checking to a program to isolate a problem such as a buffer overrun or underrun that isn't associated with a call to a memory or string function.

In the latter case, rather than keeping a pointer and performing direct manipulations on the pointer, the program may define a pointer type that contains all relevant information about the pointer, including the current value, the base pointer, and the extent of the buffer. Access to the pointer can then be controlled through macros or access functions. The access functions can perform the necessary bounds checks and print a warning message in response to attempts to exceed the bounds.

Any attempt to dereference the current pointer value can be checked against the boundaries obtained when the pointer was initialized. If the boundary is exceeded, you should call the *malloc_warning()* function to print a diagnostic message and perform error handling. The prototype is:

```
void malloc_warning( const char *funcname,
                     const char *file,
                     int line,
                     const void *link );
```

Getting pointer information

You can use these functions to obtain information about the pointer:

find_malloc_ptr()

```
void* find_malloc_ptr ( const void* ptr,
                        arena_range_t* range );
```

This function finds information about the heap buffer containing the given C pointer, including the type of allocation structure it's contained in and the pointer to the header structure for the buffer. The function returns a pointer to the **Dhead** structure associated with this particular heap buffer. You can use the returned pointer with the *DH_*()* macros to obtain more information about the heap buffer. If the pointer doesn't point into the range of a valid heap buffer, the function returns NULL.

For example, you can use the result from *find_malloc_ptr()* as an argument to *DH_ULEN()* to find out the size that the program requested for the heap buffer in the call to *malloc()*, *calloc()*, or a subsequent call to *realloc()*.

_mptr()

`char* _mptr ( const char* ptr );`

Return a pointer to the beginning of the heap buffer containing the given C pointer. You can get information about the size of the heap buffer by calling *_msize()* or *_musize()* with the value returned from this call.

Getting the heap buffer size

To obtain information about the size of a heap buffer, use the following functions:

_msize()

`ssize_t _msize( const char* ptr );`

Return the actual size of the heap buffer, given the pointer to the beginning of it. The value returned by this function is the actual size of the buffer, as opposed to the program-requested size for the buffer. The pointer must point to the beginning of the buffer — as in the case of the value returned by *_mptr()* — in order for this function to work.

_musize()

`ssize_t _musize( const char* ptr );`

Return the program-requested size of the heap buffer given the pointer to the beginning of the heap buffer. The value returned by this function is the size argument that was given to the routine that allocated the block, or to a subsequent invocation of *realloc()* that caused the block to grow.

DH_ULEN()

`DH_ULEN( ptr )`

Return the program-requested size of the heap buffer given a pointer to the **Dhead** structure, as returned by a call to *find_malloc_ptr()*. This is a macro that performs the appropriate cast on the pointer argument.

Memory leaks

The ability of the **malloc** library to keep full allocation chains of all the heap memory allocated by the program — as opposed to just accounting for some heap buffers — allows heap memory leaks to be detected by the library in response to requests by the program. Leaks can be detected in the program by performing tracing on the entire heap. This is described in the sections that follow.

Tracing

Tracing is an operation that attempts to determine whether a heap object is reachable by the program. In order to be reachable, a heap buffer must be available either directly or indirectly from a pointer in a global variable or on the stack of one of the threads. If this isn't the case, then the heap buffer is no longer visible to the program and can't be accessed without constructing a pointer that refers to the heap buffer — presumably by obtaining it from a persistent store such as a file or a shared memory object.

The set of global variables and stack for all threads is called the *root set*. Because the root set must be stable for tracing to yield valid results, tracing requires that all threads other than the one performing the trace be suspended while the trace is performed.

Tracing operates by constructing a reachability graph of the entire heap. It begins with a *root set scan* that determines the root set comprising the initial state of the reachability graph. The roots that can be found by tracing are:

- data of the program
- uninitialized data of the program
- initialized and uninitialized data of any shared objects dynamically linked into the program
- used portion of the stacks of all active threads in the program

Once the root set scan is complete, tracing initiates a *mark* operation for each element of the root set. The mark operation looks at a node of the reachability graph, scanning the memory space represented by the node, looking for pointers into the heap. Since the program may not actually have a pointer directly to the start of the buffer — but to some interior location — and it isn't possible to know which part of the root set or a heap object actually contains a pointer, tracing utilizes specialized techniques for coping with *ambiguous roots*. The approach taken is described as a conservative pointer estimation since it assumes that any word-sized object on a word-aligned memory cell that *could* point to a heap buffer or the interior of that heap buffer actually points to the heap buffer itself.

Using conservative pointer estimation for dealing with ambiguous roots, the mark operation finds all children of a node of the reachability graph. For each child in the heap that's found, it checks to see whether the heap buffer has been marked as *referenced*. If the buffer has been marked, the operation moves on to the next child. Otherwise, the trace marks the buffer, and then recursively initiates a mark operation on that heap buffer.

The tracing operation is complete when the reachability graph has been fully traversed. At this time every heap buffer that's reachable will have been marked, as could some buffers that aren't actually reachable, due to the conservative pointer estimation. Any heap buffer that hasn't been marked is definitely unreachable, constituting a memory leak. At the end of the tracing operation, all unmarked nodes can be reported as leaks.

Causing a trace and giving results

A program can cause a trace to be performed and memory leaks to be reported by calling the *malloc_dump_unreferenced()* function provided by the library:

```
int malloc_dump_unreferenced ( int fd,
                               int detail );
```

Suspend all threads, clear the mark information for all heap buffers, perform the trace operation, and print a report of all memory leaks detected. All items are reported in memory order.

fd The file descriptor on which the report should be produced.

detail How the trace operation should deal with any heap corruption problems it encounters:

1 Any problems encountered can be treated as fatal errors. After the error encountered is printed, abort the program. No report is produced.

0 Print case errors, and a report based on whatever heap information is recoverable.

Analyzing dumps

The dump of unreferenced buffers prints out one line of information for each unreferenced buffer. The information provided for a buffer includes:

- address of the buffer
- function that was used to allocate it (*malloc()*, *calloc()*, *realloc()*)
- file that contained the allocation request, if available
- line number or return address of the call to the allocation function
- size of the allocated buffer

File and line information is available if the call to allocate the buffer was made using one of the library's debug interfaces. Otherwise, the return address of the call is reported in place of the line number. In some circumstances, no return address information is available. This usually indicates that the call was made from a function with no frame information, such as the system libraries. In such cases, the entry can usually be ignored and probably isn't a leak.

From the way tracing is performed, we can see that some leaks may escape detection and may not be reported in the output. This happens if the root set or a reachable buffer in the heap has something that looks like a pointer to the buffer.

Likewise, each reported leak should be checked against the suspected code identified by the line or call return address information. If the code in question keeps interior pointers — pointers to a location inside the buffer, rather than the start of the buffer — the trace operation will likely fail to find a reference to the buffer. In this case, the buffer may well not be a leak. In other cases, there is almost certainly a memory leak.

Compiler support

The **gcc** compiler has a feature called Mudflap that adds extra code to the compiled program to check for buffer overruns. Mudflap slows a program's performance, so you should use it while testing, and turn it off in the production version. In C++ programs, you can also use the techniques described below.

C++ issues

In place of a raw pointer, C++ programs can make use of a **CheckedPtr** template that acts as a smart pointer. The smart pointer has initializers that obtain complete information about the heap buffer on an assignment operation and initialize the current pointer position. Any attempt to dereference the pointer causes bounds checking to be performed and prints a diagnostic error in response an attempt to dereference a value beyond the bounds of the buffer. The **CheckedPtr** template is provided in the **<malloc_g/malloc.h>** header for C++ programs.

You can modify the checked pointer template provided for C++ programs to suit the needs of the program. The bounds checking performed by the checked pointer is restricted to checking the actual bounds of the heap buffer, rather than the program requested size.

For C programs it's possible to compile individual modules that obey certain rules with the C++ compiler to get the behavior of the **CheckedPtr** template. C modules obeying these rules are written to a dialect of ANSI C that can be referred to as Clean C.

Clean C

The Clean C dialect is that subset of ANSI C that is compatible with the C++ language. Writing Clean C requires imposing coding conventions to the C code that restrict use to features that are acceptable to a C++ compiler. This section provides a summary of some of the more pertinent points to be considered. It is a mostly complete but by no means exhaustive list of the rules that must be applied.

To use the C++ checked pointers, the module including all header files it includes must be compatible with the Clean C subset. All the system headers for Neutrino as well as the **<malloc_g/malloc.h>** header satisfy this requirement.

The most obvious aspect to Clean C is that it must be strict ANSI C with respect to function prototypes and declarations. The use of K&R prototypes or definitions isn't allowed in Clean C. Similarly, you can't use default types for variable and function declarations.

Another important consideration for declarations is that you must provide forward declarations when referencing an incomplete structure or union. This frequently occurs for linked data structures such as trees or lists. In this case, the forward declaration must occur before any declaration of a pointer to the object in the same or another structure or union. For example, you could declare a list node as follows:

```
struct ListNode;
struct ListNode {
   struct ListNode *next;
   void *data;
};
```

Operations on void pointers are more restrictive in C++. In particular, implicit coercions from void pointers to other types aren't allowed, including both integer types and other pointer types. You must explicitly cast void pointers to other types.

The use of **const** should be consistent with C++ usage. In particular, pointers that are declared as **const** must always be used in a compatible fashion. You can't pass **const** pointers as non-**const** arguments to functions unless you typecast the **const** away.

C++ example

Here's how you could use checked pointers in the overrun example given earlier to determine the exact source of the error:

```
typedef CheckedPtr<int> intp_t;
...
intp_t foo, p;
int i;
int opt;
opt = 1;
mallopt(MALLOC_FILLAREA, opt);
foo = (int *)malloc(10*4);
opt = M_HANDLE_ABORT;
mallopt(MALLOC_WARN, opt);
for (p = foo, i = 12; i > 0; p++, i--)
    *p = 89; /* a fatal error is generated here */
opt = M_HANDLE_IGNORE;
mallopt(MALLOC_WARN, opt);
free(foo);
```

Chapter 8

Freedom from Hardware and Platform Dependencies

In this chapter. . .

Common problems

With the advent of multiplatform support, which involves non-x86 platforms as well as peripheral chipsets across these multiple platforms, we don't want to have to write different versions of device drivers for each and every platform.

While some platform dependencies are unavoidable, let's talk about some of the things that you as a developer can do to minimize the impact. At QNX Software Systems, we've had to deal with these same issues — for example, we support the 8250 serial chip on several different types of processors. Ethernet controllers, SCSI controllers, and others are no exception.

Let's look at these problems:

- I/O space vs memory-mapped
- Big-endian vs little-endian
- alignment and structure packing
- atomic operations

I/O space vs memory-mapped

The x86 architecture has two distinct address spaces:

- 16-address-line I/O space
- 32-address-line instruction and data space

The processor asserts a hardware line to the external bus to indicate which address space is being referenced. The x86 has special instructions to deal with I/O space (e.g. **IN AL, DX** vs **MOV AL,** *address*). Common hardware design on an x86 indicates that the control ports for peripherals live in the I/O address space. On non-x86 platforms, this requirement doesn't exist — all peripheral devices are mapped into various locations within the same address space as the instruction and code memory.

Big-endian vs little-endian

Big-endian vs little-endian is another compatibility issue with various processor architectures. The issue stems from the byte ordering of multibyte constants. The x86 architecture is little-endian. For example, the hexadecimal number 0x12345678 is stored in memory as:

```
address contents
      0 0x78
      1 0x56
      2 0x34
      3 0x12
```

A big-endian processor would store the data in the following order:

```
address contents
      0 0x12
```

```
1 0x34
2 0x56
3 0x78
```

This issue is worrisome on a number of fronts:

- typecast mangling
- hardware access
- network transparency

The first and second points are closely related.

Typecast mangling

Consider the following code:

```
func ()
{
    long a = 0x12345678;
    char *p;

    p = (char *) &a;
    printf ("%02X\n", *p);
}
```

On a little-endian machine, this prints the value "**`0x78`**"; on a big-endian machine, it prints "**`0x12`**". This is one of the big (pardon the pun) reasons that structured programmers generally frown on typecasts.

Hardware access

Sometimes the hardware can present you with a conflicting choice of the "correct" size for a chunk of data. Consider a piece of hardware that has a 4 KB memory window. If the hardware brings various data structures into view with that window, it's impossible to determine *a priori* what the data size should be for a particular element of the window. Is it a 32-bit long integer? An 8-bit character? Blindly performing operations as in the above code sample will land you in trouble, because the CPU will determine what it believes to be the correct endianness, regardless of what the hardware manifests.

Network transparency

These issues are naturally compounded when heterogeneous CPUs are used in a network with messages being passed among them. If the implementor of the message-passing scheme doesn't decide up front what byte order will be used, then some form of identification needs to be done so that a machine with a different byte ordering can receive and correctly decode a message from another machine. This problem has been solved with protocols like TCP/IP, where a defined *network byte order* is always adhered to, even between homogeneous machines whose byte order differs from the network byte order.

Alignment and structure packing

On the x86 CPU, you can access any sized data object at any address (albeit some accesses are more efficient than others). On non-x86 CPUs, you can't — as a general rule, you can access only N-byte objects on an N-byte boundary. For example, to access a 4-byte **`long`** integer, it must be aligned on a 4-byte address (e.g. 0x7FBBE008). An address like 0x7FBBE009 will cause the CPU to generate a fault. (An x86 processor happily generates multiple bus cycles and gets the data anyway.)

Generally, this will not be a problem with structures defined in the header files for Neutrino, as we've taken care to ensure that the members are aligned properly. The major place that this occurs is with hardware devices that can map a window into the address space (for configuration registers, etc.), and protocols where the protocol itself presents data in an unaligned manner (e.g. CIFS/SMB protocol).

Atomic operations

One final problem that can occur with different families of processors, and SMP configurations in general, is that of atomic access to variables. Since this is so prevalent with interrupt service routines and their handler threads, we've already talked about this in the chapter on Writing an Interrupt Handler.

Solutions

Now that we've seen the problems, let's take a look at some of the solutions you can use. The following header files are shipped standard with Neutrino:

`<gulliver.h>` isolates big-endian vs little-endian issues

`<hw/inout.h>` provides input and output functions for I/O or memory address spaces

Determining endianness

The file **`<gulliver.h>`** contains macros to help resolve endian issues. The first thing you may need to know is the target system's endianness, which you can find out via the following macros:

__LITTLEENDIAN__
: defined if little-endian

__BIGENDIAN__
: defined if big-endian

A common coding style in the header files (e.g. **`<gulliver.h>`**) is to check which macro is defined and to report an error if none is defined:

```
#if defined(__LITTLEENDIAN__)
// do whatever for little-endian
#elif defined(__BIGENDIAN__)
```

```
// do whatever for big-endian
#else
#error ENDIAN Not defined for system
#endif
```

The **#error** statement will cause the compiler to generate an error and abort the compilation.

Swapping data if required

Suppose you need to ensure that data obtained in the host order (i.e. whatever is “native” on this machine) is returned in a particular order, either big- or little-endian. Or vice versa: you want to convert data from host order to big- or little-endian. You can use the following macros (described here as if they’re functions for syntactic convenience):

ENDIAN_LE16()

uint16_t ENDIAN_LE16 (uint16_t *var***)**

If the host is little-endian, this macro does nothing (expands simply to *var*); else, it performs a byte swap.

ENDIAN_LE32()

uint32_t ENDIAN_LE32 (uint32_t *var***)**

If the host is little-endian, this macro does nothing (expands simply to *var*); else, it performs a quadruple byte swap.

ENDIAN_LE64()

uint64_t ENDIAN_LE64 (uint64_t *var***)**

If the host is little-endian, this macro does nothing (expands simply to *var*); else, it swaps octets of bytes.

ENDIAN_BE16()

uint16_t ENDIAN_BE16 (uint16_t *var***)**

If the host is big-endian, this macro does nothing (expands simply to *var*); else, it performs a byte swap.

ENDIAN_BE32()

uint32_t ENDIAN_BE32 (uint32_t *var***)**

If the host is big-endian, this macro does nothing (expands simply to *var*); else, it performs a quadruple byte swap.

ENDIAN_BE64()

uint64_t ENDIAN_BE64 (uint64_t *var***)**

If the host is big-endian, this macro does nothing (expands simply to *var*); else, it swaps octets of bytes.

Accessing unaligned data

To access data on nonaligned boundaries, you have to access the data one byte at a time (the correct endian order is preserved during byte access). The following macros (documented as functions for convenience) accomplish this:

UNALIGNED_RET16()

uint16_t UNALIGNED_RET16 (uint16_t ****addr16*)**

Returns a 16-bit quantity from the address specified by *addr16*.

UNALIGNED_RET32()

uint32_t UNALIGNED_RET32 (uint32_t ****addr32*)**

Returns a 32-bit quantity from the address specified by *addr32*.

UNALIGNED_RET64()

uint64_t UNALIGNED_RET64 (uint64_t ****addr64*)**

Returns a 64-bit quantity from the address specified by *addr64*.

UNALIGNED_PUT16()

void UNALIGNED_PUT16 (uint16_t ****addr16*, uint16_t** *val16***)**

Stores the 16-bit value *val16* into the address specified by *addr16*.

UNALIGNED_PUT32()

void UNALIGNED_PUT32 (uint32_t ****addr32*, uint32_t** *val32***)**

Stores the 32-bit value *val32* into the address specified by *addr32*.

UNALIGNED_PUT64()

void UNALIGNED_PUT64 (uint64_t ****addr64*, uint64_t** *val64***)**

Stores the 64-bit value *val64* into the address specified by *addr64*.

Examples

Here are some examples showing how to access different pieces of data using the macros introduced so far.

Mixed-endian accesses

This code is written to be portable. It accesses *little_data* (i.e. data that's known to be stored in little-endian format, perhaps as a result of some on-media storage scheme), and then manipulates it, writing the data back. This illustrates that the *ENDIAN_*()* macros are bidirectional.

```
uint16_t    native_data;
uint16_t    little_data;

native_data = ENDIAN_LE16 (little_data);// used as "from little-endian"
```

```
native_data++;                     // do something with native form
little_data = ENDIAN_LE16 (native_data);// used as "to little-endian"
```

Accessing hardware with dual-ported memory

Hardware devices with dual-ported memory may "pack" their respective fields on nonaligned boundaries. For example, if we had a piece of hardware with the following layout, we'd have a problem:

Address	Size	Name
`0x18000000`	1	PKTTYPE
`0x18000001`	4	PKTCRC
`0x18000005`	2	PKTLEN

Let's see why.

The first field, PKTTYPE, is fine — it's a 1-byte field, which according to the rules could be located anywhere. But the second and third fields aren't fine. The second field, PKTCRC, is a 4-byte object, but it's *not* located on a 4-byte boundary (the address is not evenly divisible by 4). The third field, PKTLEN, suffers from a similar problem — it's a 2-byte field that's not on a 2-byte boundary.

The *ideal* solution would be for the hardware manufacturer to obey the same alignment rules that are present on the target processor, but this isn't always possible. For example, if the hardware presented a raw data buffer at certain memory locations, the hardware would have no idea how you wish to interpret the bytes present — it would simply manifest them in memory.

To access these fields, you'd make a set of manifest constants for their offsets:

```
#define PKTTYPE_OFF     0x0000
#define PKTCRC_OFF      0x0001
#define PKTLEN_OFF      0x0005
```

Then, you'd map the memory region via *mmap_device_memory()*. Let's say it gave you a **`char *`** pointer called *ptr*. Using this pointer, you'd be tempted to:

```
cr1 = *(ptr + PKTTYPE_OFF);
// wrong!
sr1 = * (uint32_t *) (ptr + PKTCRC_OFF);
er1 = * (uint16_t *) (ptr + PKTLEN_OFF);
```

However, this would give you an alignment fault on non-x86 processors for the *sr1* and *er1* lines.

One solution would be to manually assemble the data from the hardware, byte by byte. And that's exactly what the *UNALIGNED_*()* macros do. Here's the rewritten example:

```
cr1 = *(ptr + PKTTYPE_OFF);
// correct!
sr1 = UNALIGNED_RET32 (ptr + PKTCRC_OFF);
er1 = UNALIGNED_RET16 (ptr + PKTLEN_OFF);
```

The access for *cr1* didn't change, because it was already an 8-bit variable — these are *always* "aligned." However, the access for the 16- and 32-bit variables now uses the macros.

An implementation trick used here is to make the pointer that serves as the base for the mapped area by a **char *** — this lets us do pointer math on it.

To write to the hardware, you'd again use macros, but this time the *UNALIGNED_PUT*()* versions:

```
*(ptr + PKTTYPE_OFF) = cr1;
UNALIGNED_PUT32 (ptr + PKTCRC_OFF, sr1);
UNALIGNED_PUT16 (ptr + PKTLEN_OFF, er1);
```

Of course, if you're writing code that should be portable to different-endian processors, you'll want to combine the above tricks with the previous endian macros. Let's define the hardware as big-endian. In this example, we've decided that we're going to store everything that the program uses in host order and do translations whenever we touch the hardware:

```
cr1 = *(ptr + PKTTYPE_OFF);  // endian neutral
sr1 = ENDIAN_BE32 (UNALIGNED_RET32 (ptr + PKTCRC_OFF));
er1 = ENDIAN_BE16 (UNALIGNED_RET16 (ptr + PKTLEN_OFF));
```

And:

```
*(ptr + PKTTYPE_OFF) = cr1;  // endian neutral
UNALIGNED_PUT32 (ptr + PKTCRC_OFF, ENDIAN_BE32 (sr1));
UNALIGNED_PUT16 (ptr + PKTLEN_OFF, ENDIAN_BE16 (er1));
```

Here's a simple way to remember which *ENDIAN_*()* macro to use. Recall that the *ENDIAN_*()* macros won't change the data on their respective platforms (i.e. the **LE** macro will return the data unchanged on a little-endian platform, and the **BE** macro will return the data unchanged on a big-endian platform). Therefore, to access the data (which we know has a *defined* endianness), we effectively want to select the *same macro as the type of data*. This way, if the platform is the same as the type of data present, no changes will occur (which is what we expect).

Accessing I/O ports

When porting code that accesses hardware, the x86 architecture has a set of instructions that manipulate a separate address space called the *I/O address space*. This address space is completely separate from the memory address space. On non-x86 platforms (PPC, etc.), such an address space doesn't exist — all devices are mapped into memory.

In order to keep code portable, we've defined a number of functions that isolate this behavior. By including the file **<hw/inout.h>**, you get the following functions:

in8()
: Reads an 8-bit value.

in16(), inbe16(), inle16()
: Reads a 16-bit value.

in32(), inbe32(), inle32()
: Reads a 32-bit value.

in8s()
: Reads a number of 8-bit values.

in16s()
: Reads a number of 16-bit values.

in32s()
: Reads a number of 32-bit values.

out8()
: Writes a 8-bit value.

out16(), outbe16(), outle16()
: Writes a 16-bit value.

out32(), outbe32(), outle32()
: Writes a 32-bit value.

out8s()
: Writes a number of 8-bit values.

out16s()
: Writes a number of 16-bit values.

out32s()
: Writes a number of 32-bit values.

On the x86 architecture, these functions perform the machine instructions **in**, **out**, **rep ins***, and **rep outs***. On non-x86 architectures, they dereference the supplied address (the *addr* parameter) and perform memory accesses.

The bottom line is that code written for the x86 will be portable to MIPS and PPC. Consider the following fragment:

```
iir = in8 (baseport);
if (iir & 0x01) {
    return;
}
```

On an x86 platform, this will perform **IN AL, DX**, whereas on a MIPS or PPC, it will dereference the 8-bit value stored at location *baseport*.

Note that the calling process must use *mmap_device_io()* to access the device's I/O registers.

Chapter 9

Conventions for Recursive Makefiles and Directories

In this chapter...

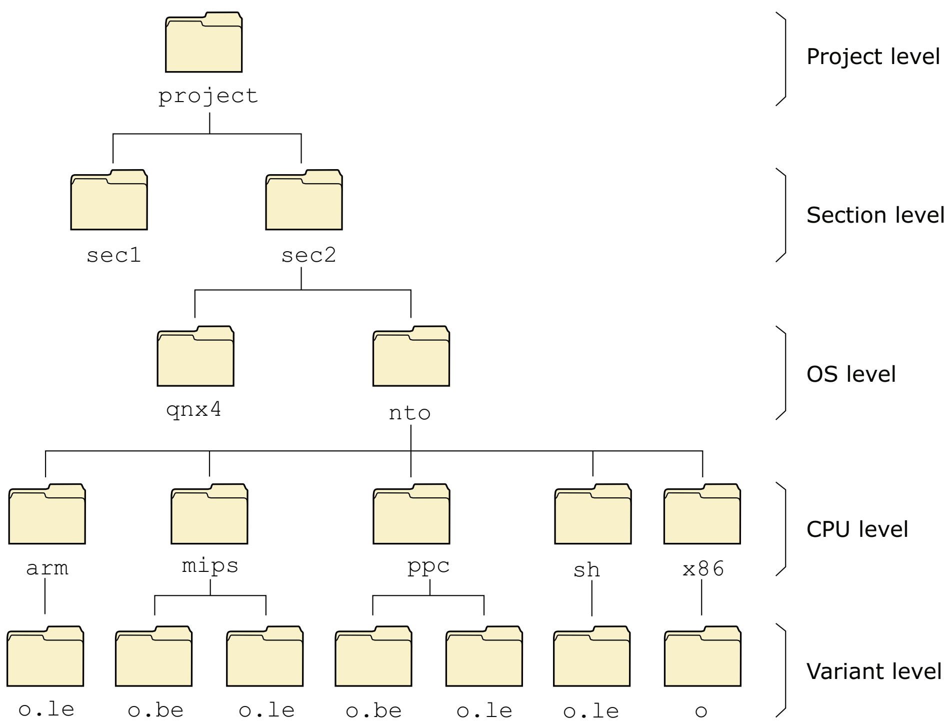
In this chapter, we'll take a look at the supplementary files used in the Neutrino development environment. Although we use the standard **make** command to create libraries and executables, you'll notice we use some of our own conventions in the **Makefile** syntax.

We'll start with a general description of a full, multiplatform source tree. Then we'll look at how you can build a tree for your products. Next. we'll discuss some advanced topics, including collapsing unnecessary levels and performing partial builds. Finally, we'll wrap up with some examples of creating **Makefile**s.

Although you're certainly not obliged to use our format for the directory structure and related tools, you may choose to use it because it's convenient for developing multiplatform code. If you do use this structure, you should use the **addvariant** command to create it; for more information, see the *Utilities Reference* as well as the examples at the end of this chapter.

Structure of a multiplatform source tree

Here's a sample directory tree for a product that can be built for two different operating systems (QNX 4 and Neutrino), on five CPU platforms (x86, MIPS, PowerPC, ARM, and SH4), with both endian combinations on the MIPS and PowerPC:



Source tree for a multiplatform project.

We'll talk about the names of the directory levels shortly. At each directory level is a **Makefile** file that the **make** utility uses to determine what to do in order to make the final executable.

However, if you examine the makefiles, you can see that most of them simply contain:

```
include recurse.mk
```

Why do we have makefiles at every level? Because **make** can recurse into the bottommost directory level (the variant level in the diagram). That's where the actual work of building the product occurs. This means that you could type **make** at the topmost directory, and it would go into all the subdirectories and compile everything. Or you could type **make** from a particular point in the tree, and it would compile only what's needed from that point down.

We'll discuss how to cause **make** to compile only certain parts of the source tree, even if invoked from the top of the tree, in the "Advanced topics" section.

When deciding where to place source files, as a rule of thumb you should place them as high up in the directory tree as possible. This not only reduces the number of directory levels to traverse when looking for source, but also encourages you to develop source that's as generic as possible (i.e. that isn't specific to the OS, CPU, or board). Lower directory levels are reserved for more and more specific pieces of source code.

If you look at the source tree that we ship, you'll notice that we follow the directory structure defined above, but with a few shortcuts. We'll cover those shortcuts in the "Advanced Topics" section.

Makefile structure

As mentioned earlier, the makefile structure is almost identical, regardless of the level that the makefile is found in. All makefiles (except the bottommost level) include the **recurse.mk** file and may set one or more macros.

Here's an example of one of our standard (nonbottommost) **Makefile**s:

```
LATE_DIRS=boards
include recurse.mk
```

The `recurse.mk` file

The **recurse.mk** file resides under **$QNX_TARGET/usr/include/mk**. This directory contains other files that are included within makefiles. Note that while the **make** utility automatically searches **$QNX_TARGET/usr/include**, we've created symbolic links from there to **$QNX_TARGET/usr/include/mk**.

The **recurse.mk** include file is typically used by higher-level makefiles to recurse into lower-level makefiles. All subdirectories present are scanned for files called **makefile** or **Makefile**. Any subdirectories that contain such files are recursed into, then **make** is invoked from within those directories, and so on, down the directory tree.

You can create a special file named **Makefile.dnm** ("dnm" stands for "Do Not Make") next to a real **Makefile** to cause **recurse.mk** *not* to descend into that directory. The contents of **Makefile.dnm** aren't examined in any way — you can use **touch** to create an empty file for it.

Macros

The example given above uses the LATE_DIRS macro. Here are the macros that you can place within a makefile:

- EARLY_DIRS
- LATE_DIRS
- LIST
- MAKEFILE
- CHECKFORCE

The EARLY_DIRS and LATE_DIRS macros

To give you some control over the ordering of the directories, the macros EARLY_DIRS and LATE_DIRS specify directories to recurse into *before* or *after* all others. You'd use this facility with directory trees that contain one directory that depends on another directory at the same level; you want the independent directory to be done first, followed by the dependent directory.

In our example above, we've specified a LATE_DIRS value of **boards**, because the **boards** directory depends on the library directory (**lib**).

Note that the EARLY_DIRS and LATE_DIRS macros accept a list of directories. The list is treated as a group, with no defined ordering *within* that group.

The LIST macro

The LIST macro serves as a tag for the particular directory level that the makefile is found in.

The LIST macro can contain a list of names that are separated by spaces. This is used when we squash directory levels together; see "Advanced Topics," later in this chapter.

Here are the common values corresponding to the directory levels:

- VARIANT
- CPU
- OS

Note that you're free to define whatever values you wish — these are simply conventions that we've adopted for the three directory levels specified. See the section on "More uses for LIST," below.

Once the directory has been identified via a tag in the makefile, you can specifically exclude or include the directory and its descendants in a **make** invocation. See "Performing partial builds," below.

The MAKEFILE macro

The MAKEFILE macro specifies the name of the makefile that **recurse.mk** should search for in the child directories. Normally this is **[Mm]akefile**, but you can set it to anything you wish by changing the MAKEFILE macro. For example, in a GNU **configure**-style makefile, **addvariant** sets it to **GNUmakefile** (see "GNU **configure**," later in this chapter.

The CHECKFORCE macro

The CHECKFORCE macro is a trigger. Its actual value is unimportant, but if you set it, the **recurse.mk** file looks for **Makefile.force** files in the subdirectories. If it finds one, **make** recurses into that directory, even if the LIST macro settings would normally prevent this from happening.

Directory levels

Let's look at the directory levels themselves in some detail. Note that you can add as many levels as you want *above* the levels described here; these levels reflect the structure of your product. For example, in a factory automation system, the product would consist of the *entire* system, and you'd then have several subdirectories under that directory level to describe various projects within that product (e.g. **gui**, **pidloop**, **robot_plc**, etc.).

Project level

You use the project-level directory mainly to store the bulk of the source code and other directories, structuring these directories logically around the project being developed. For our factory-automation example, a particular project level might be the **gui** directory, which would contain the source code for the graphical user interface as well as further subdirectories.

Section level (optional)

You use the section-level directory to contain the source base relevant to a part of the project. You can omit it if you don't need it; see "Collapsing unnecessary directory levels," later in this chapter.

OS level

If you were building products to run on multiple operating systems, you'd include an OS-level directory structure. This would serve as a branchpoint for OS-specific subdirectories. In our factory-floor example, the **gui** section might be built for both QNX 4 and Neutrino, whereas the other sections might be built just for Neutrino.

If no OS level is detected, Neutrino is assumed.

CPU level

Since we're building executables and libraries for multiple platforms, we need a place to serve as a branchpoint for the different CPUs. Generally, the CPU level contains nothing but subdirectories for the various CPUs, but it may also contain CPU-specific source files.

Variant level

Finally, the variant level contains object, library, or executable files specific to a particular variant of the processor. For example, a MIPS processor could operate in big-endian or little-endian mode. In that case, we'd have to generate two different sets of output modules. On the other hand, an x86 processor is a little-endian machine only, so we need to build only one set of output modules.

Specifying options

At the project level, there's a file called **`common.mk`**. This file contains any special flags and settings that need to be in effect in order to compile and link.

At the bottommost level (the variant level), the format of the makefile is different — it *doesn't* include **`recurse.mk`**, but instead includes **`common.mk`** (from the project level).

The `common.mk` file

The **`common.mk`** include file is where you put the traditional makefile options, such as compiler options.

In order for the **`common.mk`** file to be able to determine which system to build the particular objects, libraries, or executables for, we analyze the pathname components in the bottommost level *in reverse order* as follows:

- the last component is assigned to the VARIANT1 macro
- the next previous component is assigned to the CPU macro
- the next previous component is assigned to the OS macro
- the next previous component is assigned to the SECTION macro
- the next previous component is assigned to the PROJECT macro

For example, if we have a pathname of **`/source/factory/robot_plc/driver/nto/ppc/o.be`**, then the macros are set as follows:

Macro	Value
VARIANT1	**o.be**
CPU	**ppc**
OS	**nto**
SECTION	**driver**
PROJECT	**robot_plc**

The variant-level makefile

The variant-level makefile (i.e. the bottommost makefile in the tree) contains the single line:

```
include ../../common.mk
```

The number of **../** components must be correct to get at the **common.mk** include file, which resides in the project level of the tree. The reason that the number of **../** components isn't necessarily the same in all cases has to do with whether directory levels are being collapsed.

Recognized variant names

You can combine variant names into a *compound variant*, using a period (**.**), dash (**-**), or slash (**/**) between the variants.

The common makefiles are triggered by a number of distinguished variant names:

a The image being built is an object library.

so The image being built is a shared object.

dll The image being built is a DLL; it's linked with the **-Bsymbolic** option (see **ld** in the *Utilities Reference*).

If the compound variant doesn't include **a**, **so**, or **dll**, an executable is being built.

shared Compile the object files for **.so** use, but don't create an actual shared object. You typically use this name in an **a.shared** variant to create a static link archive that can be linked into a shared object.

g Compile and link the source with the debugging flag set.

be, **le** Compile and link the source to generate big- (if **be**) or little- (if **le**) endian code.

gcc Use the GCC (**gcc**) compiler to compile the source. If you don't specify a compiler, the makefiles provide a default.

o This is the NULL variant name. It's used when building an image that doesn't really have any variant components to it (e.g. an executable for an x86 CPU, which doesn't support bi-endian operation).

Variant names can be placed in any order in the compound variant, but to avoid confusing a source configuration management tool (e.g. CVS), make sure that the last variant in the list never looks like a generated file suffix. In other words, don't use variant names ending in **.a**, **.so**, or **.o**.

The following table lists some examples:

Variant	Purpose
g.le	A debugging version of a little-endian executable.
so.be	A big-endian version of a shared object.
403.be	A user-defined "403" variant for a big-endian system.

The only valid characters for variant names are letters, digits, and underscores (_).

In order for the source code to tell what variant(s) it's being compiled for, the common makefiles arrange for each variant name to be suffixed to the string VARIANT_ and have that defined as a C or assembler macro on the command line. For example, if the compound variant is **so.403.be**, the makefiles define the following C macros:

- VARIANT_so
- VARIANT_403
- VARIANT_be

Note that neither VARIANT_be nor VARIANT_le is defined on a CPU that doesn't support bi-endian operation, so any endian-specific code should always test for the C macros __LITTLEENDIAN__ or __BIGENDIAN__ (instead of VARIANT_le or VARIANT_be) to determine what endian-ness it's running under.

Using the standard macros and include files

We've described the pieces you'll provide when building your system, including the **common.mk** include file. Now let's look at some other include files:

- **qconfig.mk**
- **qrules.mk**
- **qtargets.mk**

We'll also look at some of the macros that these files set or use.

The `qconfig.mk` include file

Since the common makefiles have a lot of defaults based on the names of various directories, you can simplify your life enormously in the **common.mk** include file if you choose your directory names to match what the common makefiles want. For example, if the name of the project directory is the same as the name of the image, you don't have to set the NAME macro in **common.mk**.

The prototypical **common.mk** file looks like this:

```
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)

# Preset make macros go here

include $(MKFILES_ROOT)/qtargets.mk

# Post-set make macros go here
```

The **qconfig.mk** include file provides the root paths to various install, and usage trees on the system, along with macros that define the compilers and some utility commands that the makefiles use. The purpose of the **qconfig.mk** include file is to let you tailor the root directories, compilers, and commands used at your site, if they differ from the standard ones that we use and ship. Therefore, nothing in a project's makefiles should refer to a compiler name, absolute path, or command name directly. Always use the **qconfig.mk** macros.

The **qconfig.mk** file resides in **$QNX_TARGET/usr/include/mk** as **qconf-*os*.mk** (where *os* is the host OS, e.g. **nto**, **qnx4**, **linux**, **win32**), which is a symbolic link from the place where **make** wants to find it (namely **$QNX_TARGET/usr/include/qconfig.mk**). You can override the location of the include file by specifying a value for the QCONFIG macro.

If you wish to override the values of some of the macros defined in **qconfig.mk** without modifying the contents of the file, set the **QCONF_OVERRIDE** environment variable (or **make** macro) to be the name of a file to include at the end of the main **qconfig.mk** file.

> Some examples of override files set VERSION_REL, which specifies the version of QNX Neutrino that you're building for. This variable is primarily for internal use at QNX Software Systems; it indicates that **make** is running on a build machine instead of on a developer's desktop. If you set this variable, **make** becomes a lot more particular about other settings (e.g. it will insist that you set PINFO).

Preset macros

Before including **qtargets.mk**, you might need to set some macros to specify things like what additional libraries need to be searched in the link, the name of the image (if it doesn't match the project directory name), and so on. Do this in the area tagged as "**Preset make macros go here**" in the sample above.

Postset macros

Following the inclusion of **qtargets.mk**, you can override or (more likely) add to the macros set by **qtargets.mk**. Do this in the area tagged as "**Post-set make macros go here**" in the sample above.

qconfig.mk macros

Here's a summary of the macros available from **qconfig.mk**:

CP_HOST	Copy files from one spot to another.
LN_HOST	Create a symbolic link from one file to another.
RM_HOST	Remove files from the filesystem.
TOUCH_HOST	Update a file's access and modification times.
PWD_HOST	Print the full path of the current working directory.
CL_*which*	Compile and link.
CC_*which*	Compile C/C++ source to an object file.
AS_*which*	Assemble something to an object file.
AR_*which*	Generate an object file library (archive).
LR_*which*	Link a list of objects/libraries to a relocatable object file.
LD_*which*	Link a list of objects/libraries to a executable/shared object.
UM_*which*	Add a usage message to an executable.

The *which* parameter can be either the string HOST for compiling something for the host system or a triplet of the form *os_cpu_compiler* to specify a combination of target OS and CPU, as well as the compiler to be used.

The *os* is usually the string **nto** to indicate Neutrino. The *cpu* is one of **x86**, **mips**, **ppc**, **arm** or **sh**. Finally, the compiler is usually **gcc**.

For example, you could use the macro CC_nto_x86_gcc to specify:

- the compilation tool
- a Neutrino target system
- an x86 platform
- the GNU GCC compiler

The following macro contains the command-line sequence required to invoke the GCC compiler:

```
CC_nto_x86_gcc = qcc -Vgcc_ntox86 -c
```

The various makefiles use the CP_HOST, LN_HOST, RM_HOST, TOUCH_HOST, and PWD_HOST macros to decouple the OS commands from the commands used to perform the given actions. For example, under most POSIX systems, the CP_HOST macro expands to the **cp** utility. Under other operating systems, it may expand to something else (e.g. **copy**).

In addition to the macros mentioned above, you can use the following macros to specify options to be placed at the end of the corresponding command lines:

- CLPOST_*which*
- CCPOST_*which*
- ASPOST_*which*
- ARPOST_*which*
- LRPOST_*which*
- LDPOST_*which*
- UMPOST_*which*

The parameter "*which*" is the same as defined above: either the string "HOST" or the ordered triplet defining the OS, CPU, and compiler.

For example, specifying the following:

```
CCPOST_nto_x86_gcc = -ansi
```

causes the command line specified by **CC_nto_x86_gcc** to have the additional string "**-ansi**" appended to it.

The qrules.mk include file

The **qrules.mk** include file defines the macros used for compiling.

You can inspect — and in some cases, also set — the following macros when you use **qrules.mk**. Since the **qtargets.mk** file includes **qrules.mk**, these are available there as well. Don't modify those that are marked "(read-only)."

VARIANT_LIST (read-only)
: A space-separated list of the variant names macro. Useful with the **$(filter ...) make** function for picking out individual variant names.

CPU
: The name of the target CPU. Defaults to the name of the next directory up with all parent directories stripped off.

CPU_ROOT (read-only)
: The full pathname of the directory tree up to and including the OS level.

OS
: The name of the target OS. Defaults to the name of the directory two levels up with all parent directories stripped off.

OS_ROOT (read-only)
: The full pathname of the directory tree up to and including the OS level.

SECTION
: The name of the section. This is set only if there's a section level in the tree.

SECTION_ROOT (read-only)
: The full pathname of the directory tree up to and including the section level.

PROJECT (read-only)
: The *basename()* of the directory containing the **common.mk** file.

PROJECT_ROOT (read-only)
: The full pathname of the directory tree up to and including the project level.

PRODUCT (read-only)
: The *basename()* of the directory above the project level.

PRODUCT_ROOT (read-only)
: The full pathname of the directory tree up to and including the product level.

NAME
: The *basename()* of the executable or library being built. Defaults to $(PROJECT).

SRCVPATH
: A space-separated list of directories to search for source files. Defaults to all the directories from the current working directory up to and including the project root directory. You'd almost never want to set this; use EXTRA_SRCVPATH to add paths instead.

EXTRA_SRCVPATH
: Added to the end of SRCVPATH. Defaults to none.

INCVPATH
: A space-separated list of directories to search for include files. Defaults to $(SRCVPATH) plus $(USE_ROOT_INCLUDE). You'd almost never want to set this; use EXTRA_INCVPATH to add paths instead.

EXTRA_INCVPATH
: Added to INCVPATH just before the $(USE_ROOT_INCLUDE). Default is none.

LIBVPATH
: A space-separated list of directories to search for library files. Defaults to:

```
. $(INSTALL_ROOT_support)/$(OS)/$(CPUDIR)/lib $(USE_ROOT_LIB).
```

You'll almost never want to use this; use EXTRA_LIBVPATH to add paths instead.

EXTRA_LIBVPATH
: Added to LIBVPATH just before $(INSTALL_ROOT_support)**/**$(OS)**/**$(CPUDIR)**/lib**. Default is none.

DEFFILE
: The name of an assembler define file created by **mkasmoff**. Default is none.

SRCS
: A space-separated list of source files to be compiled. Defaults to all ***.s**, ***.S**, ***.c**, and ***.cc** files in SRCVPATH.

EXCLUDE_OBJS
: A space-separated list of object files *not* to be included in the link/archive step. Defaults to none.

EXTRA_OBJS
: A space-separated list of object files to be added to the link/archive step even though they don't have corresponding source files (or have been excluded by EXCLUDE_OBJS). Default is none.

OBJPREF_*object*, OBJPOST_*object*
: Options to add before or after the specified object:

OBJPREF_*object* = *options*
OBJPOST_*object* = *options*

The *options* string is inserted verbatim. Here's an example:

```
OBJPREF_libc_cut.a = -Wl,--whole-archive
OBJPOST_libc_cut.a = -Wl,--no-whole-archive
```

LIBS
: A space-separated list of library stems to be included in the link. Default is none.

LIBPREF_*library*, LIBPOST_*library*
: Options to add before or after the specified library:

LIBPREF_*library* = *options*
LIBPOST_*library* = *options*

The *options* string is inserted verbatim.

You can use these macros to link some libraries statically and others dynamically. For example, here's how to bind **libmystat.a** and **libmydyn.so** to the same program:

```
LIBS += mystat mydyn

LIBPREF_mystat = -Bstatic
LIBPOST_mystat = -Bdynamic
```

This places the **-Bstatic** option just before **-lmystat**, and **-Bdynamic** right after it, so that only that library is linked statically.

CCFLAGS
: Flags to add to the C compiler command line.

ASFLAGS
: Flags to add to the assembler command line.

LDFLAGS
: Flags to add to the linker command line.

VFLAG_*which*
: Flags to add to the command line for C compiles, assemblies, and links; see below.

CCVFLAG_*which*
: Flags to add to C compiles; see below.

ASVFLAG_*which*
: Flags to add to assemblies; see below.

LDVFLAG_*which*
: Flags to add to links; see below.

OPTIMIZE_TYPE
: The optimization type; one of:

- OPTIMIZE_TYPE=TIME — optimize for execution speed
- OPTIMIZE_TYPE=SIZE — optimize for executable size (the default)
- OPTIMIZE_TYPE=NONE — turn off optimization

Note that for the VFLAG_*which*, CCVFLAG_*which*, ASVFLAG_*which*, and LDVFLAG_*which* macros, the *which* part is the name of a variant. This combined macro is passed to the appropriate command line. For example, if there were a variant called "403," then the macro VFLAG_403 would be passed to the C compiler, assembler, and linker.

Don't use this mechanism to define a C macro constant that you can test in the source code to see if you're in a particular variant. The makefiles do that automatically for you. Don't set the *VFLAG_* macros for any of the distinguished variant names (listed in the "Recognized variant names" section, above). The common makefiles will get confused if you do.

The `qtargets.mk` include file

The **qtargets.mk** include file has the linking and installation rules.

You can inspect and/or set the following macros when you use **qtargets.mk**:

INSTALLDIR
: The subdirectory where the executable or library is to be installed. Defaults to **bin** for executables, and **lib/dll** for DLLs. If you set it to **/dev/null**, then no installation is done.

USEFILE
: The file containing the usage message for the application. Defaults to none for archives and shared objects and to $(PROJECT_ROOT)/$(NAME)**.use** for executables. The application-specific makefile can set the macro to a null string, in which case nothing is added to the executable.

LINKS
: A space-separated list of symbolic link names that are aliases for the image being installed. They're placed in the same directory as the image. The default is none.

PRE_TARGET, POST_TARGET
: Extra steps to do before/after the main target.

PRE_CLEAN, POST_CLEAN
: Extra steps to do before/after the **clean** target.

PRE_ICLEAN, POST_ICLEAN
: Extra steps to do before/after the **iclean** target.

PRE_HINSTALL, POST_HINSTALL
: Extra steps to do before/after the **hinstall** target.

PRE_CINSTALL, POST_CINSTALL
: Extra steps to do before/after the **cinstall** target.

PRE_INSTALL, POST_INSTALL
: Extra steps to do before/after the **install** target.

PRE_BUILD, POST_BUILD
: Extra steps to do before/after building the image.

SO_VERSION
: The SONAME version number to use when building a shared object (the default is 1).

PINFO
: Information to go into the ***.pinfo** file.

  For example, you can use the PINFO NAME option to to keep a permanent record of the original filename of a binary. If you use this option, the name that you specify appears in the information from the **use -i** *filename* command. Otherwise, the information from **use -i** contains the NAME entry specified outside of the PINFO define.

  For more information about PINFO, see the *hook_pinfo()* function described below for the GNU **configure** command.

Advanced topics

In this section, we'll discuss how to:

- collapse unnecessary directory levels
- perform partial builds
- perform parallel builds
- use GNU **configure**

Collapsing unnecessary directory levels

The directory structure shown in the "Structure of a multiplatform source tree" section earlier in this chapter defines the complete tree — every possible directory level is shown. In the real world, however, some of these directory levels aren't required. For example, you may wish to build a particular module for a PowerPC in little-endian mode and *never* need to build it for anything else (perhaps due to hardware constraints). Therefore, it seems a waste to have a variant level that has only the directory **o.le** and a CPU level that has only the directory **ppc**.

In this situation, you can *collapse* unnecessary directory components out of the tree. You do this by simply separating the name of the components with dashes (**-**) rather than slashes (**/**).

For example, in our source tree, let's look at the **startup/boards/800fads/ppc-be** makefile:

```
include ../common.mk
```

In this case, we've specified both the variant (as "**be**" for big-endian) and the CPU (as "**ppc**" for PowerPC) with a single directory.

Why did we do this? Because the **800fads** directory refers to a very specific board — it's not going to be useful for anything other than a PowerPC running in big-endian mode.

In this case, the makefile macros would have the following values:

Macro	Value
VARIANT1	**ppc-be**
CPU	**ppc**
OS	**nto** (default)
SECTION	**800fads**
PROJECT	**boards**

The **addvariant** command knows how to create both the squashed and unsquashed versions of the directory tree. You should always use it when creating the OS, CPU, and variant levels of the tree.

Performing partial builds

By using the LIST tag in the makefile, you can cause the **make** command to perform a partial build, even if you're at the top of the source tree.

If you were to simply type **make** without having used the LIST tag, all directories would be recursed into and everything would be built.

However, by defining a macro on **make**'s command line, you can:

- recurse into only the specified tagged directories

  Or:

- recurse into all of the directories except for the specified tagged ones

Let's consider an example. The following (issued from the top of the source tree):

```
make CPULIST=x86
```

causes only the directories that are at the CPU level and below (and tagged as LIST=CPU), *and that are called x86*, to be recursed into.

You can specify a space-separated list of directories (note the use of quoting in the shell to capture the space character):

```
make "CPULIST=x86 ppc"
```

This causes the x86 *and* PPC versions to be built.

There's also the inverse form, which causes the specific lists *not* to be built:

```
make EXCLUDE_CPULIST=ppc
```

This causes everything *except* the PowerPC versions to be built.

As you can see from the above examples, the following are all related to each other via the CPU portion:

- LIST=CPU
- CPULIST
- EXCLUDE_CPULIST

Performing parallel builds

To get **make** to run more than one command in parallel, use the JLEVEL macro. For example:

```
JLEVEL=4
```

The default value is 1. If you run parallel builds, the output from different jobs can be interspersed.

For more information, see the **-j** option in the GNU documentation for **make**.

More uses for LIST

Besides using the standard LIST values that we use, you can also define your own. Therefore, in certain makefiles, you'd put the following definition:

```
LIST=CONTROL
```

Then you can decide to build (or prevent from building) various subcomponents marked with CONTROL. This might be useful in a very big project, where compilation times are long and you need to test only a particular subsection, even though other subsections may be affected and would ordinarily be made.

For example, if you had marked two directories, **robot_plc** and **pidloop**, with the LIST=CONTROL macro within the makefile, you could then make just the **robot_plc** module:

```
make CONTROLLIST=robot_plc
```

Or make both (note the use of quoting in the shell to capture the space character):

```
make "CONTROLLIST=robot_plc pidloop"
```

Or make everything *except* the **robot_plc** module:

```
make EXCLUDE_CONTROLLIST=robot_plc
```

Or make only the **robot_plc** module for MIPS big-endian:

```
make CONTROLLIST=robot_plc CPULIST=mips VARIANTLIST=be
```

GNU configure

The **addvariant** utility knows how to work with code that uses a GNU **./configure** script for configuration. If the current working directory contains files named **configure** and **Makefile.in**, **addvariant** automatically squashes the directory levels (as described earlier) into a single OS-CPU-VARIANT level and creates **GNUmakefile** files in the newly created directories along with a recursing **Makefile**.

After you've run **addvariant**, create an executable shell script called **build-hooks** in the root of the project. This file needs to define one or more of the following shell functions (described in more detail below):

- *hook_preconfigure()*
- *hook_postconfigure()*
- *hook_premake()*
- *hook_postmake()*
- *hook_pinfo()*

Every time that you type **make** in one of the newly created directories, the **GNUmakefile** is read (a small trick that works only with GNU **make**). **GNUmakefile** in turn invokes the **$QNX_TARGET/usr/include/mk/build-cfg** script, which notices whether or not **configure** has been run in the directory:

- If it hasn't, **build-cfg** invokes the *hook_preconfigure()* function, then the project's **configure**, and then the *hook_postconfigure()* function.
- If the configure has already been done, or we just did it successfully, **build-cfg** invokes the *hook_premake()*, then does a **make -fMakefile**, then *hook_postmake()*, then *hook_pinfo()*.

If a function isn't defined in **build-hooks**, **build-cfg** doesn't bother trying to invoke it.

Within the **build-hooks** script, the following variables are available:

SYSNAME
: The host OS (e.g. **nto**, **linux**) that we're running on. This is automatically set by **build-cfg**, based on the results of **uname**.

TARGET_SYSNAME
: The target OS (e.g. **nto**, **win32**) that we're going to be generating executables for. It's set automatically by **build-cfg**, based on the directory that you're in.

make_CC
: This variable is used to set the CC **make** variable when we invoke **make**. This typically sets the compiler that **make** uses. It's set automatically by **build-cfg**, based on the directory that you're in.

make_opts
: Any additional options that you want to pass to **make** (the default is **""**).

make_cmds
: The command goals passed to **make** (e.g. **all**). It's set automatically by **build-cfg** what you passed on the original **make** command line.

configure_opts
: The list of options that should be passed to **configure**. The default is **""**, but **--srcdir=..** is automatically added just before **configure** is called.

hook_preconfigure()

This function is invoked just before we run the project's **configure** script. Its main job is to set the configure_opts variable properly. Here's a fairly complicated example (this is from GCC):

```
# The "target" variable is the compilation target: "ntoarm", "ntox86", etc.
function hook_preconfigure {
    case ${SYSNAME} in
    nto)
        case "${target}" in
        nto*)   basedir=/usr ;;
```

```
            *)      basedir=/opt/QNXsdk/host/qnx6/x86/usr ;;
            esac
            ;;
        linux)
            host_cpu=$(uname -p)
            case ${host_cpu} in
            i[34567]86) host_cpu=x86 ;;
            esac
            basedir=/opt/QNXsdk/host/linux/${host_cpu}/usr
            ;;
        *)
            echo "Don't have config for ${SYSNAME}"
            exit 1
            ;;
        esac
        configure_opts="${configure_opts} --target=${target}"
        configure_opts="${configure_opts} --prefix=${basedir}"
        configure_opts="${configure_opts} --exec-prefix=${basedir}"
        configure_opts="${configure_opts} --with-local-prefix=${basedir}"
        configure_opts="${configure_opts} --enable-haifa"
        configure_opts="${configure_opts} --enable-languages=c++"
        configure_opts="${configure_opts} --enable-threads=posix"
        configure_opts="${configure_opts} --with-gnu-as"
        configure_opts="${configure_opts} --with-gnu-ld"
        configure_opts="${configure_opts} --with-as=${basedir}/bin/${target}-as"
        configure_opts="${configure_opts} --with-ld=${basedir}/bin/${target}-ld"
        if [ ${SYSNAME} == nto ]; then
            configure_opts="${configure_opts} --enable-multilib"
            configure_opts="${configure_opts} --enable-shared"
        else
            configure_opts="${configure_opts} --disable-multilib"
        fi
    }
```

hook_postconfigure()

This is invoked after **configure** has been successfully run. Usually you don't need to define this function, but sometimes you just can't quite convince **configure** to do the right thing, so you can put some hacks in here to fix things appropriately. For example, again from GCC:

```
function hook_postconfigure {
    echo "s/^GCC_CFLAGS *=/&-I\$\(QNX_TARGET\)\/usr\/include /"  >/tmp/fix.$$
    if [ ${SYSNAME} == nto ]; then
        echo "s/OLDCC = cc/OLDCC = .\/xgcc -B.\/ -I \$\(QNX_TARGET\)\/usr\/include/" >>/tmp/fix.$$
        echo "/^INCLUDES = /s/\$/ -I\$\(QNX_TARGET\)\/usr\/include/" >>/tmp/fix.$$
        if [ ${target} == ntosh ]; then
            # We've set up GCC to support both big and little endian, but
            # we only actually support little endian right now. This will
            # cause the configures for the target libraries to fail, since
            # it will test the compiler by attempting a big endian compile
            # which won't link due to a missing libc & crt?.o files.
            # Hack things by forcing compiles/links to always be little endian
            sed -e "s/^CFLAGS_FOR_TARGET *=/&-ml /"  <Makefile >1.$$
            mv 1.$$ Makefile
        fi
    else
        # Only need to build libstdc++ & friends on one host
        rm -Rf ${target}

        echo "s/OLDCC = cc/OLDCC = .\/xgcc -B.\//" >>/tmp/fix.$$
    fi
    cd gcc
    sed -f/tmp/fix.$$ <Makefile >1.$$
    mv 1.$$ Makefile
    cd ..
    rm /tmp/fix.$$
}
```

hook_premake()

This function is invoked just before the **make**. You don't usually need it.

hook_postmake()

This function is invoked just after the **make**. We haven't found a use for this one yet, but included it for completeness.

hook_pinfo()

This function is invoked after *hook_postmake()*. Theoretically, we don't need this hook at all and we could do all its work in *hook_postmake()*, but we're keeping it separate in case we get fancier in the future.

This function is responsible for generating all the ***.pinfo** files in the project. It does this by invoking the *gen_pinfo()* function that's defined in **build-cfg**, which generates one **.pinfo**. The command line for *gen_pinfo()* is:

gen_pinfo [-n*src_name* **]** *install_name install_dir pinfo_line***...**

The arguments are:

src_name
: The name of the pinfo file (minus the **.pinfo** suffix). If it's not specified, *gen_pinfo()* uses *install_name*.

install_name
: The basename of the executable when it's installed.

install_dir
: The directory the executable should be installed in. If it doesn't begin with a **/**, the target CPU directory is prepended to it. For example, if *install_dir* is **usr/bin** and you're generating an x86 executable, the true installation directory is **/x86/usr/bin**.

pinfo_line
: Any additional pinfo lines that you want to add. You can repeat this argument as many times as required. Favorites include:

- **DESCRIPTION="This executable performs no useful purpose"**
- **SYMLINK=foobar.so**

Here's an example from the **nasm** project:

```
function hook_pinfo {
    gen_pinfo nasm    usr/bin LIC=NASM DESCRIPTION="Netwide X86 Assembler"
    gen_pinfo ndisasm usr/bin LIC=NASM DESCRIPTION="Netwide X86 Disassembler"
}
```

Examples of creating Makefiles

As mentioned earlier, if you use our directory structure, you should use the **addvariant** command to create it. This section gives some examples of creating **Makefile**s for a single application, as well as for a library and an application.

A single application

Suppose we have a product (we'll use the archiver, **lha** for this example) that we'd like to make available on all the processors that QNX Neutrino supports. Unfortunately, we've been using our own custom Makefiles for **gcc** on x86, and we have no idea how to make binaries for other processors.

The QNX Neutrino **Makefile** system makes it very easy for us to build different processor versions. Instead of writing the entire complicated **Makefile** ourselves, we simply include various QNX Neutrino **Makefile** system files that will do most of the work for us. We just have to make sure the correct variables are defined, and variants are created.

First, let's get the source code for **lha** from **http://www2m.biglobe.ne.jp/~dolphin/lha/prog/lha-114i.tar.gz** and unarchive it:

```
tar -zxvf lha-114i.tar.gz
```

This creates a directory called **lha-114i**. If we run **make** here, everything will compile, and when it's done, we'll have a x86 binary called **lha** in the **src** directory.

A typical compile command for this application using the original **Makefile** looks like:

```
gcc -O2 -DSUPPORT_LH7 -DMKSTEMP -DNEED_INCREMENTAL_INDICATOR \
  -DTMP_FILENAME_TEMPLATE=""/tmp/lhXXXXXX"" \
  -DSYSTIME_HAS_NO_TM -DEUC -DSYSV_SYSTEM_DIR -DMKTIME \
  -c -o lharc.o lharc.c
```

We want to make sure our version compiles with the same options as the original version, so we have to make sure to include those compiler options somewhere.

Let's save the current **Makefile** as a backup:

```
cd src
mv Makefile Makefile.old
```

As mentioned above, we need to define some variables in a file somewhere that our **Makefile**s can include. The usual place to put these defines is in a **common.mk** file. We can use the **addvariant** utility to create our initial **common.mk** and new **Makefile**, like this:

```
addvariant -i OS
```

Let's go through the **common.mk** file line by line to figure out what's going on, and what we need to add:

```
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)
```

You should never change these four lines. The default **qconfig.mk** defines a number of variables that the **Makefile** system uses.

After these lines, we can define our own variables. Let's start with:

```
INSTALLDIR=usr/bin
```

This defines where to install our binary. Third-party applications should go into **usr/bin** instead of **bin**, which is the default.

Next, we put in some packager info:

```
define PINFO
PINFO DESCRIPTION=Archiver using lha compression.
endef
```

If we define PINFO information like this in our **common.mk** file, a **lha.pinfo** file will be created in each of our variant directories. We'll look at this later.

After that, we add:

```
NAME=lha
```

This tells the **Makefile** system what the name of our project is. Since we're building binary executables, this will be the name of our binary.

```
#EXTRA_INCVPATH=$(PROJECT_ROOT)/includes
```

EXTRA_INCVPATH defines where our header files are located. By default, all the directories from our PROJECT_ROOT down to our variant directory are added to the main include paths (i.e. where it will look for header files.) In our case, all the project headers are located in the project's root directory, so we don't need an EXTRA_INCVPATH line. This commented-out line serves as an example.

```
EXCLUDE_OBJS=lhdir.o makezero.o
```

Ordinarily, all the source code files in the PROJECT_ROOT directory are compiled and linked to the final executable. If we want to exclude certain files from being compiled and linked, we specify the object files in EXCLUDE_OBJS.

```
CCFLAGS=-O2 -DSUPPORT_LH7 -DMKSTEMP -DNEED_INCREMENTAL_INDICATOR
  -DTMP_FILENAME_TEMPLATE=""/tmp/lhXXXXXX"" -DSYSTIME_HAS_NO_TM
  -DEUC -DSYSV_SYSTEM_DIR -DMKTIME
```

CCFLAGS defines the compiler flags. This is where we put the original compiler flags listed above.

That's all we need to add to get up and running. The last line in our **common.mk** file is:

```
include $(MKFILES_ROOT)/qtargets.mk
```

This does all the magic that figures out which CPU compiler to use, what binary to make, etc. You should never change this line.

Here's what our complete **common.mk** file looks like:

```
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)

INSTALLDIR=usr/bin
define PINFO
PINFO DESCRIPTION=Archiver using lha compression.
```

```
endef
NAME=lha
#EXTRA_INCVPATH=$(PROJECT_ROOT)/includes
EXCLUDE_OBJS=lhdir.o makezero.o
CCFLAGS=-O2 -DSUPPORT_LH7 -DMKSTEMP -DNEED_INCREMENTAL_INDICATOR
  -DTMP_FILENAME_TEMPLATE=""/tmp/lhXXXXXX"" -DSYSTIME_HAS_NO_TM
  -DEUC -DSYSV_SYSTEM_DIR -DMKTIME

include $(MKFILES_ROOT)/qtargets.mk
```

That's it for the **common.mk** file. We'll see where it is included in **Makefile**s shortly. How about the **Makefile** that was just created for us? We'll very rarely have to change any of the **Makefile**s. Usually they just contain a **LIST=** line, depending on where they are in our directory tree, and some **Makefile** code to include the appropriate file that makes the recursion into subdirectories possible. The exception is the **Makefile** at the very bottom. More on this later.

We'll have to have a usage description for our application as well. In our case, we can get a usage message simply by running **lha** without any parameters, like this:

```
./lha 2> lha.use
```

For our final binaries, when someone types **use lha** (assuming **lha** is in their path), they'll get the proper usage message.

As described earlier in this appendix, we use a lot of subdirectories. Here are the ones we need:

Directory	Level
nto	OS
nto/x86/	CPU
nto/x86/o	Variant

Unless we'll be releasing for QNX 4 as well as Neutrino, we'll need only the **nto** directory for the OS level. For the CPU level, we'll have directories for anything we want to support: ARM, SH, PPC, MIPS, and/or x86.

The final variant directory depends on what we're building, and what endian-ness we want to compile for:

- Since x86 only has little endian-ness, it doesn't have an extension.
- If there's a choice, the variant level directory name would have a **.be** or **.le** at the end (e.g. **o.le**).
- If we're building shared libraries, we'd replace the **o** variant with a **so** variant.
- If we were building shared objects that aren't meant to be linked directly with applications, we'd use a **dll** variant.
- If we were building static libraries, we'd use an **a** variant.

We're building just an executable binary, so we use the **o** variant. Each directory and subdirectory needs to have a **Makefile**. Again, for most of them we're simply including the **recurse.mk** file, which contains everything needed to recurse down our tree until we get to the **o*** directory, as well as setting a LIST variable, which for general use indicates where we are in our **Makefile** tree. For example, if the directory contains variants, LIST is set to **VARIANT**.

Let's use the **addvariant** utility to create a directory tree and appropriate **Makefile**s for our various CPUs and variants. The **addvariant** utility can do more than just add variants, but in our case, that's all we need it for. We create a variant by running:

```
addvariant nto
```

Let's do this for each of our CPUs, like this:

```
addvariant nto mips o.le
addvariant nto mips o.be
addvariant nto arm o.le
addvariant nto sh o.le
addvariant nto ppc o.be
addvariant nto x86 o
```

If we look at the **Makefile** in the **lha-114i/src/nto/x86/o** directory, we see it just contains:

```
include ../../../common.mk
```

Since this is the bottom directory, we don't need to recurse any further, but rather we want to include the **common.mk** file we created earlier. We also don't need a LIST variable, since we have no subdirectories at this point. We now have a complete QNX Neutrino-style **Makefile** tree.

A library and an application

What if we want to distribute shared libraries (again for all the CPUs we can) and a development package as well? Let's use the **bzip2** distribution as our example. The **bzip2** binary already comes with QNX Neutrino, but the library doesn't. You can download the source code from **http://www.bzip.org**.

This is a good example, because it contains both a library (**libbz2**) and an application (**bzip2**). Once we've downloaded it, we can extract and build it with:

```
tar -zxvf bzip2-1.0.3.tar.gz
cd bzip2-1.0.3
make
```

We notice that a typical compile looks like:

```
gcc -Wall -Winline -O2 -fomit-frame-pointer -fno-strength-reduce
  -D_FILE_OFFSET_BITS=64 -c decompress.c
```

Let's remember those options for later.

The problem with using the QNX Neutrino makefile system in this case, is that we want to make two projects: the **libbz2** library and the **bzip2** application. With the QNX Neutrino **Makefile** system, we usually have a single project.

The best solution is to separate them into different directories. Instead of moving the source code around, we'll just create two subdirectories, one called **lib**, and the other called **app**, in which we'll create the appropriate **Makefile** and **common.mk** files:

```
mkdir app
cd app
addvariant -i OS
addvariant nto arm o.le
addvariant nto mips o.le
addvariant nto mips o.be
addvariant nto sh o.le
addvariant nto ppc o.be
addvariant nto x86 o
cd ..
mkdir lib
cd lib
addvariant -i OS
addvariant nto arm so.le
addvariant nto arm a.le
addvariant nto mips so.le
addvariant nto mips so.be
addvariant nto mips a.le
addvariant nto mips a.be
addvariant nto sh so.le
addvariant nto sh a.le
addvariant nto ppc so.be
addvariant nto ppc a.be
addvariant nto x86 so
addvariant nto x86 a
```

If we try to build either of these projects now, not much happens. This is because we haven't told the **Makefile** system where our source files are.

Let's start with the library. Its **common.mk** file already contains the default lines:

```
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)

include $(MKFILES_ROOT)/qtargets.mk
```

Let's add a few more lines just before the line that includes **qtargets.mk**. First, we'll add the compile options it used originally:

```
CCFLAGS+=-Wall -Winline -O2 -fomit-frame-pointer -fno-strength-reduce
  -D_FILE_OFFSET_BITS=64
```

Next, let's tell it where to find the source files. The PRODUCT_ROOT directory is the parent directory of the PROJECT_ROOT directory, which is where our source code is located. Let's use that to specify where our source code is:

```
EXTRA_SRCVPATH=$(PRODUCT_ROOT)
```

Since the parent directory also contains the source code for the **bzip2** application, and we want only the object files for the **libbz** library, let's weed out the object files we don't need:

```
EXCLUDE_OBJS=bzip2recover.o bzip2.o dlltest.o spewG.o unzcrash.o
```

We should add some PINFO definitions and specify where to install the files (**usr/lib** in this case):

```
define PINFO
PINFO DESCRIPTION=bzip2 data compressions library
endef
INSTALLDIR=usr/lib
```

Finally, let's make sure the library has the correct name. By default, it uses the directory name of the PROJECT_ROOT directory. Since we don't want to call our library **lib**, let's change it:

```
NAME=bz2
```

If we now run **make** at the terminal, we can watch all of our libraries being built.

You may notice that there are **libbz2S.a** libraries being built in the **so** directories. You can use these libraries if you want to create other shared libraries that require code from this library.

What about our application, **bzip2**? Let's change into the **app** directory we built before and set up our **common.mk** file. This time, though, we exclude everything but the **bzip2.o** from our objects and add a new line:

```
LIBS+=bz2
```

Here's our complete **common.mk** file:

```
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)

CCFLAGS+=-Wall -Winline -O2 -fomit-frame-pointer -fno-strength-reduce
  -D_FILE_OFFSET_BITS=64
EXTRA_SRCVPATH=$(PRODUCT_ROOT)
EXCLUDE_OBJS= blocksort.o bzip2recover.o bzlib.o compress.o crctable.o
   decompress.o dlltest.o huffman.o randtable.o spewG.o unzcrash.o
LIBS+=bz2
define PINFO
PINFO DESCRIPTION=bzip2 file compressor/decompressor
endef
INSTALLDIR=usr/bin
NAME=bzip2

include $(MKFILES_ROOT)/qtargets.mk
```

We can easily create our **bzip2.use** file by getting help from our previously created **bzip2** executable:

```
../bzip2 --help 2> bzip2.use
```

Now we can build our binaries, and make sure they exist:

```
make
ls -l nto/*/*/bzip2
```

Appendix A

Using GDB

In this appendix. . .

Neutrino-specific extensions

The Neutrino implementation of GDB includes some extensions:

`target qnx`
: Set the target; see "Setting the target."

`set nto-inherit-env`
: Set where the remote process inherits its environment from; see "Your program's environment."

`set nto-cwd`
: Set the working directory for the remote process; see "Starting your program."

`set nto-timeout`
: Set the timeout for remote reads; see "Setting the target."

`upload` *local_path remote_path*
: Send a file to a remote target system.

`download` *remote_path local_path*
: Retrieve a file from a remote target system.

`info pidlist`
: Display a list of processes and their process IDs on the remote system

`info meminfo`
: Display a list of memory-region mappings (shared objects) for the current process being debugged.

A quick overview of starting the debugger

To debug an application, do the following:

1. Start GDB, but don't specify the application as an argument:

   ```
   gdb
   ```

2. Load the symbol information for the application:

   ```
   file my_application
   ```

3. If you're debugging remotely, set the target:

   `target qnx` *com_port_specifier* | *host:port* | **`pty`**

4. If you're debugging remotely, send the application to the target:

   ```
   upload my_application /tmp/my_application
   ```

5. Set any breakpoints. For example, to set a breakpoint in *main()*:

```
set break main
```

6 Start the application:

```
run
```

GDB commands

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous; and you can repeat certain GDB commands by typing just Enter. You can also use the Tab key to get GDB to fill out the rest of a word in a command (or to show you the alternatives available, if there's more than one possibility).

You may also place GDB commands in an initialization file and these commands will be run before any that have been entered via the command line. For more information, see:

- **gdb** in the *Utilities Reference*
- the GNU documentation for GDB

Command syntax

A GDB command is a single line of input. There's no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command **step** accepts an argument that is the number of times to step, as in **step 5**. You can also use the **step** command with no arguments. Some command names don't allow any arguments.

GDB command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, **s** is specifically defined as equivalent to **step** even though there are other commands whose names start with **s**. You can test abbreviations by using them as arguments to the **help** command.

A blank line as input to GDB (typing just Enter) means to repeat the previous command. Certain commands (for example, **run**) don't repeat this way; these are commands whose unintentional repetition might cause trouble and which you're unlikely to want to repeat.

When you repeat the **list** and **x** commands with Enter, they construct new arguments rather than repeat exactly as typed. This permits easy scanning of source or memory.

GDB can also use Enter in another way: to partition lengthy output, in a way similar to the common utility **more**. Since it's easy to press one Enter too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a **#** to the end of the line is a comment. This is useful mainly in command files.

Command completion

GDB can fill in the rest of a word in a command for you if there's only one possibility; it can also show you what the valid possibilities are for the next word in a command, at any time. This works for GDB commands, GDB subcommands, and the names of symbols in your program.

Press the Tab key whenever you want GDB to fill out the rest of a word. If there's only one possibility, GDB fills in the word, and waits for you to finish the command (or press Enter to enter it). For example, if you type:

```
(gdb) info bre Tab
```

GDB fills in the rest of the word **breakpoints**, since that is the only **info** subcommand beginning with **bre**:

```
(gdb) info breakpoints
```

You can either press Enter at this point, to run the **info breakpoints** command, or backspace and enter something else, if **breakpoints** doesn't look like the command you expected. (If you were sure you wanted **info breakpoints** in the first place, you might as well just type Enter immediately after **info bre**, to exploit command abbreviations rather than command completion).

If there's more than one possibility for the next word when you press Tab, GDB sounds a bell. You can either supply more characters and try again, or just press Tab a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with **make_**, but when you type:

```
b make_Tab
```

GDB just sounds the bell. Typing Tab again displays all the function names in your program that begin with those characters, for example:

```
make_a_section_from_file     make_environ
make_abs_section             make_function_type
make_blockvector             make_pointer_type
make_cleanup                 make_reference_type
make_command                 make_symbol_completion_list
(gdb) b make_
```

After displaying the available possibilities, GDB copies your partial input (**b make_** in the example) so you can finish the command.

If you just want to see the list of alternatives in the first place, you can press Esc followed by ? (rather than press Tab twice).

Sometimes the string you need, while logically a "word", may contain parentheses or other characters that GDB normally excludes from its notion of a word. To permit word completion to work in this situation, you may enclose words in **'** (single quote marks) in GDB commands.

The most likely situation where you might need this is in typing the name of a C++ function. This is because C++ allows function overloading (multiple definitions of the same function, distinguished by argument type). For example, when you want to set a breakpoint you may need to distinguish whether you mean the version of **name** that takes an **int** parameter, **name(int)**, or the version that takes a **float** parameter, **name(float)**. To use the word-completion facilities in this situation, type a single quote **'** at the beginning of the function name. This alerts GDB that it may need to consider more information than usual when you press Tab, or Esc followed by ?, to request word completion:

```
(gdb) b 'bubble(Esc?
bubble(double,double)    bubble(int,int)
(gdb) b 'bubble(
```

In some cases, GDB can tell that completing a name requires using quotes. When this happens, GDB inserts the quote for you (while completing as much as it can) if you don't type the quote in the first place:

```
(gdb) b bub Tab
```

GDB alters your input line to the following, and rings a bell:

```
(gdb) b 'bubble(
```

In general, GDB can tell that a quote is needed (and inserts it) if you haven't yet started typing the argument list when you ask for completion on an overloaded symbol.

Getting help

You can always ask GDB itself for information on its commands, using the command **help**.

help
h
: You can use **help** (**h**) with no arguments to display a short list of named classes of commands:

```
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain
points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features

Type "help" followed by a class name for a list
of commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if
unambiguous.
(gdb)
```

help *class*
: Using one of the general help classes as an argument, you can get a list of the individual commands in that class. For example, here's the help display for the class **status**:

```
(gdb) help status
Status inquiries.

List of commands:

show -- Generic command for showing things set
with "set"
info -- Generic command for printing status

Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if
unambiguous.
(gdb)
```

help *command*
: With a command name as **help** argument, GDB displays a short paragraph on how to use that command.

complete *args*
: The **complete** *args* command lists all the possible completions for the beginning of a command. Use *args* to specify the beginning of the command you want completed. For example:

```
complete i
```

results in:

```
info
inspect
ignore
```

This is intended for use by GNU Emacs.

In addition to **help**, you can use the GDB commands **info** and **show** to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under **info** and **show** in the index point to all the sub-commands.

info
: This command (abbreviated **i**) is for describing the state of your program. For example, you can list the arguments given to your program with **info args**, list the registers currently in use with **info registers**, or list the breakpoints you've set with **info breakpoints**. You can get a complete list of the **info** sub-commands with **help info**.

set
: You can assign the result of an expression to an environment variable with **set**. For example, you can set the GDB prompt to a $-sign with **set prompt $**.

`show` In contrast to **`info`**, **`show`** is for describing the state of GDB itself. You can change most of the things you can **`show`**, by using the related command **`set`**; for example, you can control what number system is used for displays with **`set radix`**, or simply inquire which is currently in use with **`show radix`**.

To display all the settable parameters and their current values, you can use **`show`** with no arguments; you may also use **`info set`**. Both commands produce the same display.

Here are three miscellaneous **`show`** subcommands, all of which are exceptional in lacking corresponding **`set`** commands:

`show version` Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may occasionally want to determine which version of GDB you're running; as GDB evolves, new commands are introduced, and old ones may wither away. The version number is also announced when you start GDB.

`show copying` Display information about permission for copying GDB.

`show warranty` Display the GNU "NO WARRANTY" statement.

Running programs under GDB

To run a program under GDB, you must first generate debugging information when you compile it. You may start GDB with its arguments, if any, in an environment of your choice. You may redirect your program's input and output, debug an already running process, or kill the process being debugged.

Compiling for debugging

Debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the **`-g`** option when you run the compiler.

GCC, the GNU C compiler, supports **`-g`** with or without **`-O`**, making it possible to debug optimized code. We recommend that you *always* use **`-g`** whenever you compile a program. You may think your program is correct, but there's no sense in pushing your luck.

When you debug a program compiled with **`-g -O`**, remember that the optimizer is rearranging your code; the debugger shows you what is really there. Don't be too surprised when the execution path doesn't exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable—because the compiler optimizes it out of existence.

Some things don't work as well with `-g -O` as with just `-g`, particularly on machines with instruction scheduling. If in doubt, recompile with `-g` alone, and if this fixes the problem, please report it to us — and include a test case.

Setting the target

If you're debugging locally, you don't need to specify the target (or you can specify **`target procfs`**).

If you're debugging remotely, you need to specify the target to use:

`target qnx` *com_port_specifier* | *host:port* | **`pty`**

The **`pty`** option spawns a **`pdebug`** server on the local machine and connects via a pty.

The **`devc-pty`** manager must be running on the machine that's running **`pdebug`**, and a ptyp/ttyp pair must be available.

Starting your program

`set nto-cwd` *path*

Specify the remote process's working directory. You should do this before starting your program.

`run`
`r`

Use the **`run`** command to start your program under GDB. You must first specify the program name with an argument to GDB (see the description of the **`gdb`** utility).

The **`run`** creates an inferior process and makes that process run your program.

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do *before* starting your program. (You can change it after starting your program, but such changes affect your program the *next* time you start it.) This information may be divided into the following categories:

Arguments
: Specify the arguments to give your program as the arguments of the **`run`** command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the **SHELL** environment variable. See "Your program's arguments/"

Environment
: Your program normally inherits its environment from GDB, but you can use the GDB commands **`set environment`** and **`unset environment`** to change parts of the environment that affect your program. See "Your program's environment."

While input and output redirection work, you can't use pipes to pass the output of the program you're debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table and reads it again. When it does this, GDB tries to retain your current breakpoints.

Here's an example of starting a program for local debugging:

```
(gdb) file /tmp/helloworld
Reading symbols from /tmp/helloworld...done.
(gdb) b main
Breakpoint 1 at 0x804860c: file ./main.c, line 5.
(gdb) r
Starting program:
Remote: /tmp/helloworld

Breakpoint 1, main () at ./main.c:5
5       {
(gdb)
```

Here's an example of starting the program for remote debugging:

```
(gdb) target qnx mytst:8000
Remote debugging using mytst:8000
Remote target is little-endian
(gdb) file /tmp/helloworld
Reading symbols from /tmp/helloworld...done.
(gdb) upload /tmp/helloworld /tmp/helloworld
(gdb) b main
Breakpoint 1 at 0x804860c: file ./main.c, line 5.
(gdb) r
Starting program:
Remote: /tmp/helloworld

Breakpoint 1, main () at ./main.c:5
5       {
(gdb)
```

If your communication line is slow, you might need to set the timeout for remote reads:

set nto-timeout *time*

where *time* is the timeout, in seconds. The default is 10 seconds.

Your program's arguments

The arguments to your program can be specified by the arguments of the **run** command.

A **run** command with no arguments uses the same arguments used by the previous **run**, or those set by the **set args** command.

set args
: Specify the arguments to be used the next time your program is run. If **set args** has no arguments, **run** executes your program with no

arguments. Once you've run your program with arguments, using **set args** before the next **run** is the only way to run it again without arguments.

show args Show the arguments to give your program when it's started.

Your program's environment

The *environment* consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they're inherited by all the other programs you run. When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

set nto-inherit-env *value*

If *value* is 0, the process inherits its environment from GDB. If *value* is 1 (the default), the process inherits its environment from **pdebug**.

path *directory* Add *directory* to the front of the **PATH** environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by a colon (**:**) or whitespace. If *directory* is already in the path, it's moved to the front, so it's searched sooner.

You can use the string **$cwd** to refer to the current working directory at the time GDB searches the path. A period (**.**) refers to the directory where you executed the **path** command. GDB replaces the period in the *directory* argument by the current path before adding *directory* to the search path.

show paths Display the list of search paths for executables (the **PATH** environment variable).

show environment [*varname*]

Print the value of environment variable *varname* to be given to your program when it starts. If you don't supply *varname*, print the names and values of all environment variables to be given to your program. You can abbreviate **environment** as **env**.

set environment *varname* [=] *value*

Set environment variable *varname* to *value*. The value changes for your program only, not for GDB itself. The *value* may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The *value* parameter is optional; if it's eliminated, the variable is set to a null value.

For example, this command:

```
set env USER=foo
```

tells a Unix program, when subsequently run, that its user is named **foo**.

unset environment *varname*

Remove variable *varname* from the environment to be passed to your program. This is different from **set env** *varname* **=**, in that **unset environment** removes the variable from the environment, rather than assign it an empty value.

Your program's input and output

By default, the program you run under GDB does input and output to the same terminal that GDB uses. GDB switches the terminal to its own terminal modes to interact with you, but it records the terminal modes your program was using and switches back to them when you continue running your program.

You can redirect your program's input and/or output using shell redirection with the **run** command. For example,

```
run > outfile
```

starts your program, diverting its output to the file **outfile**.

Debugging an already-running process

attach *process-id*

This command attaches to a running process—one that was started outside GDB. (The **info files** command shows your active targets.) The command takes as its argument a process ID. To find out a process ID, use the **pidin** utility (see the *Utilities Reference*), or use GDB's **info pidlist** command.

The **attach** command doesn't repeat if you press Enter a second time after executing the command.

To use **attach**, you must have permission to send the process a signal.

When using **attach**, you should first use the **file** command to specify the program running in the process and load its symbol table.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with **run**. You can insert breakpoints; you can step and continue; you can modify storage. If you want the process to continue running, use the **continue** command after attaching GDB to the process.

detach When you've finished debugging the attached process, you can use the **detach** command to release it from GDB control. Detaching the process continues its execution. After the **detach** command, that process and GDB become completely independent once more, and you're ready to **attach** another process or start one with **run**. The **detach** command doesn't repeat if you press Enter again after executing the command.

If you exit GDB or use the **run** command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the **set confirm** command.

Killing the process being debugged

kill Kill the process being debugged.

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

The **kill** command is also useful if you wish to recompile and relink your program. With Neutrino, it's possible to modify an executable file while it's running in a process. If you want to run the new version, kill the current process; when you next type **run**, GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current breakpoint settings).

Debugging programs with multiple threads

In Neutrino, a single program may have more than one *thread* of execution. Each thread has its own registers and execution stack, and perhaps private memory.

GDB provides these facilities for debugging multithreaded programs:

- **thread** *threadno*, a command to switch between threads
- **info threads**, a command to inquire about existing threads
- **thread apply** [*threadno*] [*all*] *args*,
  a command to apply a command to a list of threads
- thread-specific breakpoints

The GDB thread debugging facility lets you observe all threads while your program runs—but whenever GDB takes control, one thread in particular is always the focus of debugging. This thread is called the *current thread*. Debugging commands show program information from the perspective of the current thread.

GDB associates its own thread number—always a single integer—with each thread in your program.

info threads Display a summary of all threads currently in your program. GDB displays for each thread (in this order):

1. Thread number assigned by GDB
2. Target system's thread identifier (*systag*)
3. Current stack frame summary for that thread.

An asterisk **`*`** to the left of the GDB thread number indicates the current thread. For example:

```
(gdb) info threads
  3 process 35 thread 27  0x34e5 in sigpause ()
  2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1, argv=0x7ffffff8)
    at threadtest.c:68
```

`thread` *threadno*
: Make thread number *threadno* the current thread. The command argument *threadno* is the internal GDB thread number, as shown in the first field of the **`info threads`** display. GDB responds by displaying the system identifier of the thread you selected and its current stack frame summary:

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

`thread apply` [*threadno*] [**`all`**] *args*
: The **`thread apply`** command lets you apply a command to one or more threads. Specify the numbers of the threads that you want affected with the command argument *threadno*. To apply a command to all threads, use **`thread apply all`** *args*.

Whenever GDB stops your program because of a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. GDB alerts you to the context switch with a message of the form **`[Switching to`** *systag***`]`** to identify the thread.

See "Stopping and starting multithreaded programs" for more information about how GDB behaves when you stop and start programs with multiple threads.

See "Setting watchpoints" for information about watchpoints in programs with multiple threads.

Debugging programs with multiple processes

GDB has no special support for debugging programs that create additional processes using the *fork()* function. When a program forks, GDB continues to debug the parent process, and the child process runs unimpeded. If you've set a breakpoint in any code that the child then executes, the child gets a **`SIGTRAP`** signal, which (unless it catches the signal) causes it to terminate.

However, if you want to debug the child process, there's a workaround that isn't too painful:

1. Put a call to *sleep()* in the code that the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay doesn't occur when you don't want to run GDB on the child.
2. While the child is sleeping, get its process ID by using the **`pidin`** utility (see the *Utilities Reference*) or by using GDB's **`info pidlist`** command.
3. Tell GDB (a new invocation of GDB if you're also debugging the parent process) to attach to the child process (see "Debugging an already-running process"). From that point on you can debug the child process just like any other process that you've attached to.

Stopping and continuing

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as **`step`**. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide ample explanation of the status of your program—but you can also explicitly request this information at any time.

`info program`
: Display information about the status of your program: whether it's running or not, what process it is, and why it stopped.

Breakpoints, watchpoints, and exceptions

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the **`break`** command and its variants (see "Setting breakpoints") to specify the place where your program should stop by line number, function name or exact address in the program. In languages with exception handling (such as GNU C++), you can also set breakpoints where an exception is raised (see "Breakpoints and exceptions").

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see "Setting watchpoints"), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See "Automatic display."

GDB assigns a number to each breakpoint or watchpoint when you create it; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on your program until you enable it again.

Setting breakpoints

Use the **break** (**b**) command to set breakpoints. The debugger convenience variable **$bpnum** records the number of the breakpoints you've set most recently; see "Convenience variables" for a discussion of what you can do with convenience variables.

You have several ways to say where the breakpoint should go:

break *function*
: Set a breakpoint at entry to *function*. When using source languages such as C++ that permit overloading of symbols, *function* may refer to more than one possible place to break. See "Breakpoint menus" for a discussion of that situation.

break +*offset*
break -*offset*
: Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

break *linenum*
: Set a breakpoint at line *linenum* in the current source file. That file is the last file whose source text was printed. This breakpoint stops your program just before it executes any of the code on that line.

break *filename*:*linenum*
: Set a breakpoint at line *linenum* in source file *filename*.

break *filename*:*function*
: Set a breakpoint at entry to *function* found in file *filename*. Specifying a filename as well as a function name is superfluous except when multiple files contain similarly named functions.

**break ****address*
: Set a breakpoint at address *address*. You can use this to set breakpoints in parts of your program that don't have debugging information or source files.

break
: When called without any arguments, **break** sets a breakpoint at the next instruction to be executed in the selected stack frame (see "Examining the Stack"). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a **finish** command in the frame inside the selected frame—except that **finish** doesn't leave an active breakpoint. If you use **break** without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops.

  GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it didn't do this, you wouldn't be able to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when your program stopped.

break ... if *cond*

Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. The ellipsis (**...**) stands for one of the possible arguments described above (or no argument) specifying where to break. For more information on breakpoint conditions, see "Break conditions."

There are several variations on the **break** command, all using the same syntax as above:

tbreak
: Set a breakpoint enabled only for one stop. The breakpoint is set in the same way as for the **break** command, except that it's automatically deleted after the first time your program stops there. See "Disabling breakpoints."

hbreak
: Set a hardware-assisted breakpoint. The breakpoint is set in the same way as for the **break** command, except that it requires hardware support (and some target hardware may not have this support).

The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction.

thbreak
: Set a hardware-assisted breakpoint enabled only for one stop. The breakpoint is set in the same way as for the **break** command. However, like the **tbreak** command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the **hbreak** command, the breakpoint requires hardware support, which some target hardware may not have. See "Disabling breakpoints" and "Break conditions."

rbreak *regex*
: Set breakpoints on all functions matching the regular expression *regex*. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they're treated just like the breakpoints set with the **break** command. You can delete them, disable them, or make them conditional the same way as any other breakpoint.

When debugging C++ programs, **rbreak** is useful for setting breakpoints on overloaded functions that aren't members of any special classes.

The following commands display information about breakpoints and watchpoints:

info breakpoints [*n*]
info break [*n*]
info watchpoints [*n*]

Print a table of all breakpoints and watchpoints set and not deleted, with the following columns for each breakpoint:

- Breakpoint Numbers.
- Type — breakpoint or watchpoint.
- Disposition — whether the breakpoint is marked to be disabled or deleted when hit.
- Enabled or Disabled — enabled breakpoints are marked with **y**, disabled with **n**.
- Address — where the breakpoint is in your program, as a memory address.
- What — where the breakpoint is in the source for your program, as a file and line number.

If a breakpoint is conditional, **info break** shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that.

An **info break** command with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable **$_** and the default examining-address for the **x** command are set to the address of the last breakpoint listed (see "Examining memory").

The **info break** command displays the number of times the breakpoint has been hit. This is especially useful in conjunction with the **ignore** command. You can ignore a large number of breakpoint hits, look at the breakpoint information to see how many times the breakpoint was hit, and then run again, ignoring one less than that number. This gets you quickly to the last hit of that breakpoint.

GDB lets you set any number of breakpoints at the same place in your program. There's nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see "Break conditions").

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of **longjmp** (in C programs). These internal breakpoints are assigned negative numbers, starting with **-1**; **info breakpoints** doesn't display them.

You can see these breakpoints with the GDB maintenance command, **maint info breakpoints**.

maint info breakpoints

Using the same format as **info breakpoints**, display both the breakpoints you've set explicitly and those GDB is using for internal purposes. The type column identifies what kind of breakpoint is shown:

- **`breakpoint`** — normal, explicitly set breakpoint.
- **`watchpoint`** — normal, explicitly set watchpoint.
- **`longjmp`** — internal breakpoint, used to handle correctly stepping through **`longjmp`** calls.
- **`longjmp resume`** — internal breakpoint at the target of a **`longjmp`**.
- **`until`** — temporary internal breakpoint used by the GDB **`until`** command.
- **`finish`** — temporary internal breakpoint used by the GDB **`finish`** command.

Setting watchpoints

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

Although watchpoints currently execute two orders of magnitude more slowly than other breakpoints, they can help catch errors where in cases where you have no clue what part of your program is the culprit.

`watch` *expr*
: Set a watchpoint for an expression. GDB breaks when *expr* is written into by the program and its value changes.

`rwatch` *arg*
: Set a watchpoint that breaks when watch *arg* is read by the program. If you use both watchpoints, both must be set with the **`rwatch`** command.

`awatch` *arg*
: Set a watchpoint that breaks when *arg* is read and written into by the program. If you use both watchpoints, both must be set with the **`awatch`** command.

`info watchpoints`
: This command prints a list of watchpoints and breakpoints; it's the same as **`info break`**.

In multithreaded programs, watchpoints have only limited usefulness. With the current watchpoint implementation, GDB can watch the value of an expression *in a single thread only*. If you're confident that the expression can change due only to the current thread's activity (and if you're also confident that no other thread can become current), then you can use watchpoints as usual. However, GDB may not notice when a noncurrent thread's activity changes the expression.

Breakpoints and exceptions

Some languages, such as GNU C++, implement exception handling. You can use GDB to examine what caused your program to raise an exception and to list the exceptions your program is prepared to handle at a given point in time.

catch *exceptions*
: You can set breakpoints at active exception handlers by using the **catch** command. The *exceptions* argument is a list of names of exceptions to catch.

You can use **info catch** to list active exception handlers. See "Information about a frame."

There are currently some limitations to exception handling in GDB:

- If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program to continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits.
- You can't raise an exception interactively.
- You can't install an exception handler interactively.

Sometimes **catch** isn't the best way to debug exception handling: if you need to know exactly where an exception is raised, it's better to stop *before* the exception handler is called, since that way you can see the stack before any unwinding takes place. If you set a breakpoint in an exception handler instead, it may not be easy to find out where the exception was raised.

To stop just before an exception handler is called, you need some knowledge of the implementation. In the case of GNU C++, exceptions are raised by calling a library function named *__raise_exception()*, which has the following ANSI C interface:

```
void __raise_exception (void **addr, void *id);

/* addr is where the exception identifier is stored.
   id is the exception identifier.  */
```

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on *__raise_exception()*. See "Breakpoints, watchpoints, and exceptions."

With a conditional breakpoint (see "Break conditions") that depends on the value of *id*, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

Deleting breakpoints

You often need to eliminate a breakpoint or watchpoint once it's done its job and you no longer want your program to stop there. This is called *deleting* the breakpoint. A breakpoint that has been deleted no longer exists and is forgotten.

With the **clear** command you can delete breakpoints according to where they are in your program. With the **delete** command you can delete individual breakpoints or watchpoints by specifying their breakpoint numbers.

You don't have to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

`clear` Delete any breakpoints at the next instruction to be executed in the selected stack frame (see "Selecting a frame"). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

`clear` *function*
`clear` *filename*:*function*

Delete any breakpoints set at entry to *function*.

`clear` *linenum*
`clear` *filename*:*linenum*

Delete any breakpoints set at or within the code of the specified line.

`delete` [**`breakpoints`**] [*bnums*...]

Delete the breakpoints or watchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks for confirmation, unless you've **`set confirm off`**). You can abbreviate this command as **`d`**.

Disabling breakpoints

Rather than delete a breakpoint or watchpoint, you might prefer to *disable* it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can *enable* it again later.

You disable and enable breakpoints and watchpoints with the **`enable`** and **`disable`** commands, optionally specifying one or more breakpoint numbers as arguments. Use **`info break`** or **`info watch`** to print a list of breakpoints or watchpoints if you don't know which numbers to use.

A breakpoint or watchpoint can have any of the following states:

Enabled The breakpoint stops your program. A breakpoint set with the **`break`** command starts out in this state.

Disabled The breakpoint has no effect on your program.

Enabled once The breakpoint stops your program, but then becomes disabled. A breakpoint set with the **`tbreak`** command starts out in this state.

Enabled for deletion

The breakpoint stops your program, but immediately afterwards it's deleted permanently.

You can use the following commands to enable or disable breakpoints and watchpoints:

`disable` [**`breakpoints`**] [*bnums*...]
: Disable the specified breakpoints—or all breakpoints, if none is listed. A disabled breakpoint has no effect but isn't forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate **`disable`** as **`dis`**.

`enable` [**`breakpoints`**] [*bnums*...]
: Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

`enable [breakpoints] once` *bnums*...
: Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately after stopping your program.

`enable [breakpoints] delete` *bnums*...
: Enable the specified breakpoints to work once, then die. GDB deletes any of these breakpoints as soon as your program stops there.

Except for a breakpoint set with **`tbreak`** (see "Setting breakpoints"), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command **`until`** can set and delete a breakpoint of its own, but it doesn't change the state of your other breakpoints; see "Continuing and stepping.")

Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language (see "Expressions"). A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is *true*.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated—that is, when the condition is false. In C, if you want to test an assertion expressed by the condition *assert*, you should set the condition **`!`** *assert* on the appropriate breakpoint.

Conditions are also accepted for watchpoints; you may not need them, since a watchpoint is inspecting the value of an expression anyhow—but it might be simpler, say, to just set a watchpoint on a variable name, and specify a condition that tests whether the new value is an interesting one.

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there's another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see "Breakpoint command lists").

Break conditions can be specified when a breakpoint is set, by using **`if`** in the arguments to the **`break`** command. See "Setting breakpoints." They can also be changed at any time with the **`condition`** command. The **`watch`** command doesn't recognize the **`if`** keyword; **`condition`** is the only way to impose a further condition on a watchpoint.

`condition` *bnum expression*
: Specify *expression* as the break condition for breakpoint or watchpoint number *bnum*. After you set a condition, breakpoint *bnum* stops your program only if the value of *expression* is true (nonzero, in C). When you use **`condition`**, GDB checks *expression* immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. GDB doesn't actually evaluate *expression* at the time the **`condition`** command is given, however. See "Expressions."

`condition` *bnum*
: Remove the condition from breakpoint number *bnum*. It becomes an ordinary unconditional breakpoint.

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there's a special way to do it, using the *ignore count* of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is *n*, the breakpoint doesn't stop the next *n* times your program reaches it.

`ignore` *bnum count*
: Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, your program's execution doesn't stop; other than to decrement the ignore count, GDB takes no action.

  To make the breakpoint stop the next time it's reached, specify a count of zero.

  When you use **`continue`** to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to **`continue`**, rather than use **`ignore`**. See "Continuing and stepping."

  If a breakpoint has a positive ignore count and a condition, the condition isn't checked. Once the ignore count reaches zero, GDB resumes checking the condition.

You could achieve the effect of the ignore count with a condition such as **$foo-- <= 0** using a debugger convenience variable that's decremented each time. See "Convenience variables."

Breakpoint command lists

You can give any breakpoint (or watchpoint) a series of commands to execute when your program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

commands [*bnum*]
... *command-list* **...**
end

Specify a list of commands for breakpoint number *bnum*. The commands themselves appear on the following lines. Type a line containing just **end** to terminate the commands.

To remove all commands from a breakpoint, type **commands** and follow it immediately with **end**; that is, give no commands.

With no *bnum* argument, **commands** refers to the last breakpoint or watchpoint set (not to the breakpoint most recently encountered).

Pressing Enter as a means of repeating the last GDB command is disabled within a *command-list*.

You can use breakpoint commands to start your program up again. Just use the **continue** command, or **step**, or any other command that resumes execution.

Commands in *command-list* that follow a command that resumes execution are ignored. This is because any time you resume execution (even with a simple **next** or **step**), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If the first command you specify in a command list is **silent**, the usual message about stopping at a breakpoint isn't printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the remaining commands print anything, you see no sign that the breakpoint was reached. The **silent** command is meaningful only at the beginning of a breakpoint command list.

The commands **echo**, **output**, and **printf** allow you to print precisely controlled output, and are often useful in silent breakpoints.

For example, here's how you could use breakpoint commands to print the value of *x* at entry to *foo()* whenever *x* is positive:

```
break foo if x>0
commands
silent
printf "x is %d\n",x
cont
end
```

One application for breakpoint commands is to compensate for one bug so you can test for another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the **continue** command so that your program doesn't stop, and start with the **silent** command so that no output is produced. Here's an example:

```
break 403
commands
silent
set x = y + 4
cont
end
```

Breakpoint menus

Some programming languages (notably C++) permit a single function name to be defined several times, for application in different contexts. This is called *overloading*. When a function name is overloaded, **break** *function* isn't enough to tell GDB where you want a breakpoint.

If you realize this is a problem, you can use something like **break** *function*(*types*) to specify which particular version of the function you want. Otherwise, GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt **>**. The first two options are always **[0] cancel** and **[1] all**. Typing 1 sets a breakpoint at each definition of *function*, and typing 0 aborts the **break** command without setting any new breakpoints.

For example, the following session excerpt shows an attempt to set a breakpoint at the overloaded symbol *String::after()*. We choose three particular definitions of that function name:

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
 breakpoints.
(gdb)
```

Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If due to a signal, you may want to use **handle**, or use **signal 0** to resume execution. See "Signals.")

continue [*ignore-count*]
c [*ignore-count*]
fg [*ignore-count*]

> Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* lets you specify a further number of times to ignore a breakpoint at this location; its effect is like that of **ignore** (see "Break conditions").
>
> The argument *ignore-count* is meaningful only when your program stopped due to a breakpoint. At other times, the argument to **continue** is ignored.
>
> The synonyms **c** and **fg** are provided purely for convenience, and have exactly the same behavior as **continue**.

To resume execution at a different place, you can use **return** (see "Returning from a function") to go back to the calling function; or **jump** (see "Continuing at a different address") to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see "Breakpoints, watchpoints, and exceptions") at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

step Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated **s**.

If you use the **step** command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it doesn't step into a function that is compiled without debugging information. To step through functions without debugging information, use the **stepi** command, described below.

The **step** command stops only at the first instruction of a source line. This prevents multiple stops in switch statements, for loops, etc. The **step** command stops if a function that has debugging information is called within the line.

Also, the **`step`** command enters a subroutine only if there's line number information for the subroutine. Otherwise it acts like the **`next`** command. This avoids problems when using **`cc -gl`** on MIPS machines.

`step` *count*
: Continue running as in **`step`**, but do so *count* times. If a breakpoint is reached, or a signal not related to stepping occurs before *count* steps, stepping stops right away.

`next` [*count*]
: Continue to the next source line in the current (innermost) stack frame. This is similar to **`step`**, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the **`next`** command. This command is abbreviated **`n`**.

The *count* argument is a repeat count, as for **`step`**.

The **`next`** command stops only at the first instruction of a source line. This prevents the multiple stops in switch statements, for loops, etc.

`finish`
: Continue running until just after function in the selected stack frame returns. Print the returned value (if any).

Contrast this with the **`return`** command (see "Returning from a function").

`u`
`until`
: Continue running until a source line past the current line in the current stack frame is reached. This command is used to avoid single-stepping through a loop more than once. It's like the **`next`** command, except that when **`until`** encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

This means that when you reach the end of a loop after single-stepping though it, **`until`** makes your program continue execution until it exits the loop. In contrast, a **`next`** command at the end of a loop simply steps back to the beginning of the loop, which forces you to step through the next iteration.

The **`until`** command always stops your program if it attempts to exit the current stack frame.

The **`until`** command may produce somewhat counterintuitive results if the order of machine code doesn't match the order of the source lines. For example, in the following excerpt from a debugging session, the **`f`** (**`frame`**) command shows that execution is stopped at line **`206`**; yet when we use **`until`**, we get to line **`195`**:

```
(gdb) f
#0  main (argc=4, argv=0xf7fffae8) at m4.c:206
206                 expand_input();
(gdb) until
195             for ( ; argc > 0; NEXTARG) {
```

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop—even though the test in a C **for**-loop is written before the body of the loop. The **until** command appeared to step back to the beginning of the loop when it advanced to this expression; however, it hasn't really gone to an earlier statement—not in terms of the actual machine code.

An **until** command with no argument works by means of single instruction stepping, and hence is slower than **until** with an argument.

until *location*
u *location*
: Continue running your program until either the specified location is reached, or the current stack frame returns. The *location* is any of the forms of argument acceptable to **break** (see "Setting breakpoints"). This form of the command uses breakpoints, and hence is quicker than **until** without an argument.

stepi [*count*]
si [*count*]
: Execute one machine instruction, then stop and return to the debugger.

It's often useful to do **display/i $pc** when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See "Automatic display."

The *count* argument is a repeat count, as in **step**.

nexti [*count*]
ni [*count*]
: Execute one machine instruction, but if it's a function call, proceed until the function returns.

The *count* argument is a repeat count, as in **next**.

Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. The table below gives several examples of signals:

Signal:	Received when:
SIGINT	You type an interrupt, Ctrl-C
SIGSEGV	The program references a place in memory far away from all the areas in use.
SIGALRM	The alarm clock timer goes off (which happens only if your program has requested an alarm).

Some signals, including **SIGALRM**, are a normal part of the functioning of your program. Others, such as **SIGSEGV**, indicate errors; these signals are *fatal* (killing your program immediately) if the program hasn't specified in advance some other way to handle the signal. **SIGINT** doesn't indicate an error in your program, but it's normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. You can tell GDB in advance what to do for each kind of signal. Normally, it's set up to:

- Ignore signals like **SIGALRM** that don't indicate an error so as not to interfere with their role in the functioning of your program.
- Stop your program immediately whenever an error signal happens.

You can change these settings with the **handle** command.

info signals
info handle
: Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

handle *signal keywords...*
: Change the way GDB handles signal *signal*. The *signal* can be the number of a signal or its name (with or without the **SIG** at the beginning). The *keywords* say what change to make.

The keywords allowed by the **handle** command can be abbreviated. Their full names are:

nostop
: GDB shouldn't stop your program when this signal happens. It may still print a message telling you that the signal has come in.

stop
: GDB should stop your program when this signal happens. This implies the **print** keyword as well.

print
: GDB should print a message when this signal happens.

noprint
: GDB shouldn't mention the occurrence of the signal at all. This implies the **nostop** keyword as well.

pass GDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled.

nopass GDB shouldn't allow your program to see this signal.

When a signal stops your program, the signal isn't visible until you continue. Your program sees the signal then, if **pass** is in effect for the signal in question *at that time*. In other words, after GDB reports a signal, you can use the **handle** command with **pass** or **nopass** to control whether your program sees that signal when you continue.

You can also use the **signal** command to prevent your program from seeing a signal, or cause it to see a signal it normally doesn't see, or to give it any signal at any time. For example, if your program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with **signal 0**. See "Giving your program a signal."

Stopping and starting multithreaded programs

When your program has multiple threads (see "Debugging programs with multiple threads"), you can choose whether to set breakpoints on all threads, or on a particular thread.

break *linespec* **thread** *threadno*
break *linespec* **thread** *threadno* **if** ...

The *linespec* specifies source lines; there are several ways of writing them, but the effect is always to specify some source line.

Use the qualifier **thread** *threadno* with a breakpoint command to specify that you want GDB to stop the program only when a particular thread reaches this breakpoint. The *threadno* is one of the numeric thread identifiers assigned by GDB, shown in the first column of the **info threads** display.

If you don't specify **thread** *threadno* when you set a breakpoint, the breakpoint applies to *all* threads of your program.

You can use the **thread** qualifier on conditional breakpoints as well; in this case, place **thread** *threadno* before the breakpoint condition, like this:

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

Whenever your program stops under GDB for any reason, *all* threads of execution stop, not just the current thread. This lets you examine the overall state of the program, including switching between threads, without worrying that things may change underfoot.

Conversely, whenever you restart the program, *all* threads start executing. *This is true even when single-stepping* with commands like **step** or **next**.

In particular, GDB can't single-step all threads in lockstep. Since thread scheduling is up to the Neutrino microkernel (not controlled by GDB), other threads may execute more than one statement while the current thread completes a single step. Moreover, in general, other threads stop in the middle of a statement, rather than at a clean statement boundary, when the program stops.

You might even find your program stopped in another thread after continuing or even single-stepping. This happens whenever some other thread runs into a breakpoint, a signal, or an exception before the first thread completes whatever you requested.

Examining the stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by GDB, and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you're interested in. See “Selecting a frame.”

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the **`frame`** command (see “Information about a frame”).

Stack frames

The call stack is divided up into contiguous pieces called *stack frames*, or *frames* for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function *main()*. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with 0 for the innermost frame, 1 for the frame that called it, and so on upward. These numbers don't really exist in your program; they're assigned by GDB to give you a way of designating stack frames in GDB commands.

Some compilers provide a way to compile functions so that they operate without stack frames. (For example, the **gcc** option **-fomit-frame-pointer** generates functions without a frame.) This is occasionally done with heavily used library functions to reduce the time required to set up the frame. GDB has limited facilities for dealing with these function invocations. If the innermost function invocation has no stack frame, GDB nevertheless regards it as though it had a separate frame, which is numbered 0 as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

frame *args*
: The **frame** command lets you move from one stack frame to another, and to print the stack frame you select. The *args* may be either the address of the frame or the stack frame number. Without an argument, **frame** prints the current stack frame.

select-frame
: The **select-frame** command lets you move from one stack frame to another without printing the frame. This is the silent version of **frame**.

Backtraces

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame 0), followed by its caller (frame 1), and on up the stack.

backtrace
bt
: Print a backtrace of the entire stack, with one line per frame, for all frames in the stack.

  You can stop the backtrace at any time by typing the system interrupt character, normally Ctrl-C.

backtrace *n*
bt *n*
: Similar, but print only the innermost *n* frames.

backtrace -*n*
bt -*n*
: Similar, but print only the outermost *n* frames.

The names **where** and **info stack** (**info s**) are additional aliases for **backtrace**.

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use **set print address off**.

The backtrace also shows the source filename and line number, as well as the arguments to the function. The program counter value is omitted if it's at the beginning of the code for that line number.

Here's an example of a backtrace. It was made with the command **bt 3**, so it shows the innermost three frames:

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
(More stack frames follow...)
```

The display for frame 0 doesn't begin with a program counter value, indicating that your program has stopped at the beginning of the code for line **993** of **builtin.c**.

Selecting a frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

frame *n*
f *n*
: Select frame number *n*. Recall that frame 0 is the innermost (currently executing) frame, frame 1 is the frame that called the innermost one, and so on. The highest-numbered frame is the one for **main**.

frame *addr*
f *addr*
: Select the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful when your program has multiple stacks and switches between them.

: On the MIPS architecture, **frame** needs two addresses: a stack pointer and a program counter.

up *n*
: Move *n* frames up the stack. For positive numbers, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. The default for *n* is 1.

down *n*
: Move *n* frames down the stack. For positive numbers, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. The default for *n* is 1. You may abbreviate **down** as **do**.

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the source file and line number of execution in that frame. The second line shows the text of that source line.

For example:

```
(gdb) up
#1  0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc)
    at env.c:10
10              read_input_file (argv[i]);
```

After such a printout, the **list** command with no arguments prints ten lines centered on the point of execution in the frame. See "Printing source lines."

up-silently *n*
down-silently *n*

These two commands are variants of **up** and **down**; they differ in that they do their work silently, without causing display of the new frame. They're intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting.

Information about a frame

There are several other commands to print information about the selected stack frame:

frame
f

When used without any argument, this command doesn't change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated **f**. With an argument, this command is used to select a stack frame. See "Selecting a frame."

info frame
info f

This command prints a verbose description of the selected stack frame, including:

- the address of the frame
- the address of the next frame down (called by this frame)
- the address of the next frame up (caller of this frame)
- the language in which the source code corresponding to this frame is written
- the address of the frame's arguments
- the program counter saved in it (the address of execution in the caller frame)
- which registers were saved in the frame

The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *addr*
info f *addr*

Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the **frame** command. See “Selecting a frame.”

info args Print the arguments of the selected frame, each on a separate line.

info locals Print the local variables of the selected frame, each on a separate line. These are all variables (declared either static or automatic) accessible at the point of execution of the selected frame.

info catch Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the **up**, **down**, or **frame** commands); then type **info catch**. See “Breakpoints and exceptions.”

MIPS machines and the function stack

MIPS-based computers use an unusual stack frame, which sometimes requires GDB to search backward in the object code to find the beginning of a function.

To improve response time — especially for embedded applications, where GDB may be restricted to a slow serial line for this search — you may want to limit the size of this search, using one of these commands:

set heuristic-fence-post *limit*

Restrict GDB to examining at most *limit* bytes in its search for the beginning of a function. A value of 0 (the default) means there’s no limit. However, except for 0, the larger the limit the more bytes **heuristic-fence-post** must search and therefore the longer it takes to run.

show heuristic-fence-post

Display the current limit.

These commands are available *only* when GDB is configured for debugging programs on MIPS processors.

Examining source files

GDB can print parts of your program’s source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see “Selecting a frame”), GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

Printing source lines

To print lines from a source file, use the **list** (**l**) command. By default, ten lines are printed. There are several ways to specify what part of the file you want to print. Here are the forms of the **list** command most commonly used:

list *linenum* Print lines centered around line number *linenum* in the current source file.

list *function* Print lines centered around the beginning of function *function*.

list Print more lines. If the last lines printed were printed with a **list** command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see “Examining the Stack”), this prints lines centered around that line.

list - Print lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the **list** command. You can change this using **set listsize**:

set listsize *count*
Make the **list** command display *count* source lines (unless the **list** argument explicitly specifies some other number).

show listsize Display the number of lines that **list** prints.

Repeating a **list** command with Enter discards the argument, so it’s equivalent to typing just **list**. This is more useful than listing the same lines again. An exception is made for an argument of **-**; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the **list** command expects you to supply zero, one or two *linespecs*. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. Here’s a complete description of the possible arguments for **list**:

list *linespec* Print lines centered around the line specified by *linespec*.

list *first*,*last* Print lines from *first* to *last*. Both arguments are linespecs.

list ,*last* Print lines ending with *last*.

list *first***,** Print lines starting with *first*.

list + Print lines just after the lines last printed.

list - Print lines just before the lines last printed.

list As described in the preceding table.

Here are the ways of specifying a single source line—all the kinds of *linespec*:

number Specifies line *number* of the current source file. When a **list** command has two linespecs, this refers to the same source file as the first linespec.

+*offset* Specifies the line *offset* lines after the last line printed. When used as the second linespec in a **list** command that has two, this specifies the line *offset* lines down from the first linespec.

-*offset* Specifies the line *offset* lines before the last line printed.

filename:*number* Specifies line *number* in the source file *filename*.

function Specifies the line that begins the body of the function *function*. For example: in C, this is the line with the open brace, **}**.

filename:*function* Specifies the line of the open brace that begins the body of *function* in the file *filename*. You need the filename with a function name only to avoid ambiguity when there are identically named functions in different source files.

*****address* Specifies the line containing the program address *address*. The *address* may be any expression.

Searching source files

The commands for searching through the current source file for a regular expression are:

forward-search *regexp*
search *regexp*
fo *regexp*

Check each line, starting with the one following the last line listed, for a match for *regexp*, listing the line found.

reverse-search *regexp*
rev *regexp*

Check each line, starting with the one before the last line listed and going backward, for a match for *regexp*, listing the line found.

Specifying source directories

Executable programs sometimes don't record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB wants a source file, it tries all the directories in the list, in the order they're present in the list, until it finds a file with the desired name.

The executable search path *isn't* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

If GDB can't find a source file in the source path, and the object program records a directory, GDB tries that directory too. If the source path is empty, and there's no record of the compilation directory, GDB looks in the current directory as a last resort.

Whenever you reset or rearrange the source path, GDB clears out any information it has cached about where source files are found and where each line is in the file.

When you start GDB, its source path is empty. To add other directories, use the **`directory`** command.

`directory` *dirname ...*
`dir` *dirname ...*

Add directory *dirname* to the front of the source path. Several directory names may be given to this command, separated by colons (**`:`**) or whitespace. You may specify a directory that is already in the source path; this moves it forward, so GDB searches it sooner.

You can use the string **`$cdir`** to refer to the compilation directory (if one is recorded), and **`$cwd`** to refer to the current working directory. Note that **`$cwd`** isn't the same as a period (**`.`**); the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

`directory`

Reset the source path to empty again. This requires confirmation.

`show directories`

Print the source path: show which directories it contains.

If your source path is cluttered with directories that are no longer of interest, GDB may sometimes cause confusion by finding the wrong versions of source. You can correct the situation as follows:

1. Use **`directory`** with no argument to reset the source path to empty.
2. Use **`directory`** with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

Source and machine code

You can use the command **info line** to map source lines to program addresses (and vice versa), and the command **disassemble** to display a range of addresses as machine instructions. When run under GNU Emacs mode, the **info line** command causes the arrow to point to the line specified. Also, **info line** prints addresses in symbolic form as well as hex.

info line *linespec*

Print the starting and ending addresses of the compiled code for source line *linespec*. You can specify source lines in any of the ways understood by the **list** command (see "Printing source lines").

For example, we can use **info line** to discover the location of the object code for the first line of function **m4_changequote**:

```
(gdb) info line m4_changecom
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can also inquire (using ******addr* as the form for *linespec*) what source line covers a particular address:

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

After **info line**, the default address for the **x** command is changed to the starting address of the line, so that **x/i** is sufficient to begin examining the machine code (see "Examining memory"). Also, this address is saved as the value of the convenience variable **$_** (see "Convenience variables").

disassemble
: This specialized command dumps a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; GDB dumps the function surrounding this value. Two arguments specify a range of addresses (first inclusive, second exclusive) to dump.

We can use **disassemble** to inspect the object code range shown in the last **info line** example (the example shows SPARC machine instructions):

```
(gdb) disas 0x63e4 0x6404
Dump of assembler code from 0x63e4 to 0x6404:
0x63e4 <builtin_init+5340>:     ble 0x63f8 <builtin_init+5360>
0x63e8 <builtin_init+5344>:     sethi %hi(0x4c00), %o0
0x63ec <builtin_init+5348>:     ld [%i1+4], %o0
0x63f0 <builtin_init+5352>:     b 0x63fc <builtin_init+5364>
0x63f4 <builtin_init+5356>:     ld [%o0+4], %o0
0x63f8 <builtin_init+5360>:     or %o0, 0x1a4, %o0
0x63fc <builtin_init+5364>:     call 0x9288 <path_search>
0x6400 <builtin_init+5368>:     nop
End of assembler dump.
```

set assembly-language *instruction-set*

This command selects the instruction set to use when disassembling the program via the **disassemble** or **x/i** commands. It's useful for architectures that have more than one native instruction set.

Currently it's defined only for the Intel x86 family. You can set *instruction-set* to either **i386** or **i8086**. The default is **i386**.

Shared libraries

You can use the following commands when working with shared libraries:

sharedlibrary [*regexp*]

Load shared object library symbols for files matching the given regular expression, *regexp*. If *regexp* is omitted, GDB tries to load symbols for all loaded shared libraries.

info sharedlibrary

Display the status of the loaded shared object libraries.

The following parameters apply to shared libraries:

set solib-search-path *dir*[:*dir*...]

Set the search path for loading shared library symbols files that don't have an absolute path. This path overrides the **PATH** and **LD_LIBRARY_PATH** environment variables.

set solib-absolute-prefix *prefix*

Set the prefix for loading absolute shared library symbol files.

set auto-solib-add *value*

Make the loading of shared library symbols automatic or manual:

- If *value* is nonzero, symbols from all shared object libraries are loaded automatically when the inferior process (i.e. the one being debugged) begins execution, or when the dynamic linker informs GDB that a new library has been loaded.
- If *value* is zero, symbols must be loaded manually with the **sharedlibrary** command.

You can query the settings of these parameters with the **show solib-search-path**, **show solib-absolute-prefix**, and **show auto-solib-add** commands.

Examining data

The usual way to examine data in your program is with the **print** (**p**) command or its synonym **inspect**. It evaluates and prints the value of an expression of the language your program is written in.

print *exp*
print /*f exp*
: *exp* is an expression (in the source language). By default, the value of *exp* is printed in a format appropriate to its data type; you can choose a different format by specifying **/***f*, where *f* is a letter specifying the format; see "Output formats."

print
print /*f*
: If you omit *exp*, GDB displays the last value again (from the *value history*; see "Value history"). This lets you conveniently inspect the same value in an alternative format.

A lower-level way of examining data is with the **x** command. It examines data in memory at a specified address and prints it in a specified format. See "Examining memory."

If you're interested in information about types, or about how the fields of a structure or class are declared, use the **ptype** *exp* command rather than **print**. See "Examining the symbol table."

Expressions

The **print** command and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you're using is valid in an expression in GDB. This includes conditional expressions, function calls, casts and string constants. It unfortunately doesn't include symbols defined by preprocessor **#define** commands.

GDB supports array constants in expressions input by the user. The syntax is {*element*, *element*...}. For example, you can use the command **print {1, 2, 3}** to build up an array in memory that is *malloc*'d in the target program.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it's useful to cast a number into a pointer in order to examine a structure at that address in memory.

GDB supports these operators, in addition to those common to programming languages:

@
: Binary operator for treating parts of memory as arrays. See "Artificial arrays", for more information.

`::`
: Lets you specify a variable in terms of the file or function where it's defined. See "Program variables."

{*type*} *addr*
: Refers to an object of type *type* stored at address *addr* in memory. The *addr* may be any expression whose value is an integer or pointer (but parentheses are required around binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at *addr*.

Program variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see "Selecting a frame"); they must be either:

- global (or static)

  Or:

- visible according to the scope rules of the programming language from the point of execution in that frame

This means that in the function:

```
foo (a)
     int a;
{
  bar (a);
  {
    int b = test ();
    bar (b);
  }
}
```

you can examine and use the variable *a* whenever your program is executing within the function *foo()*, but you can use or examine the variable *b* only while your program is executing inside the block where *b* is declared.

There's an exception: you can refer to a variable or function whose scope is a single source file even if the current execution point isn't in this file. But it's possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file, using the colon-colon notation:

file`::`*variable*
function`::`*variable*

Here *file* or *function* is the name of the context for the static *variable*. In the case of filenames, you can use quotes to make sure GDB parses the filename as a single word. For example, to print a global value of **x** defined in **f2.c**:

```
(gdb) p 'f2.c'::x
```

This use of `::` is very rarely in conflict with the very similar use of the same notation in C++. GDB also supports use of the C++ scope resolution operator in GDB expressions.

Occasionally, a local variable may appear to have the wrong value at certain points in a function, such as just after entry to a new scope, and just before exit.

You may see this problem when you're stepping by machine instructions. This is because, on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you're stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

Artificial arrays

It's often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

You can do this by referring to a contiguous span of memory as an *artificial array*, using the binary operator **@**. The left operand of **@** should be the first element of the desired array and be an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left operand. The first element is actually the left operand; the second element comes from bytes of memory immediately following those that hold the first element, and so on. For example, if a program says:

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of **array** with:

```
p *array@len
```

The left operand of **@** must reside in memory. Array values made with **@** in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see "Value history"), after printing one out.

Another way to create an artificial array is to use a cast. This reinterprets a value as if it were an array. The value need not be in memory:

```
(gdb) p/x (short[2])0x12345678
$1 = {0x1234, 0x5678}
```

As a convenience, if you leave the array length out — as in (*type*[])*value* — gdb calculates the size to fill the value as **sizeof(***value***)/sizeof(***type***)**. For example:

```
(gdb) p/x (short[])0x12345678
$2 = {0x1234, 0x5678}
```

Sometimes the artificial array mechanism isn't quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you're interested in the values of pointers in an array. One useful workaround in this situation is to use a convenience variable (see "Convenience variables") as a counter in an expression that prints the first interesting value, and then repeat that expression via Enter. For instance, suppose you have an array *dtab* of pointers to structures, and you're interested in the values of a field *fv* in each structure. Here's an example of what you might type:

```
set $i = 0
p dtab[$i++]->fv
Enter
Enter
...
```

Output formats

By default, GDB prints a value according to its data type. Sometimes this isn't what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the **print** command with a slash and a format letter. The format letters supported are:

x Regard the bits of the value as an integer, and print the integer in hexadecimal.

d Print as integer in signed decimal.

u Print as integer in unsigned decimal.

o Print as integer in octal.

t Print as integer in binary. The letter **t** stands for **two**. (The letter **b** can't be used because these format letters are also used with the **x** command, where **b** stands for **byte**. See "Examining memory.")

a Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

c Regard as an integer and print it as a character constant.

f Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see “Registers”), type:

```
p/x $pc
```

No space is required before the slash; this is because command names in GDB can’t contain a slash.

To reprint the last value in the value history with a different format, you can use the **print** command with just a format and no expression. For example, **p/x** reprints the last value in hex.

Examining memory

You can use the command **x** (for “examine”) to examine memory in any of several formats, independently of your program’s data types.

x/*nfu addr*
x *addr*
x Use the **x** command to examine memory.

The *n*, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; *addr* is an expression giving the address where you want to start displaying memory. If you use defaults for *nfu*, you need not type the slash **/**. Several commands set convenient defaults for *addr*.

n The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units *u*) to display.

f The display format is one of the formats used by **print**, **s** (null-terminated string), or **i** (machine instruction). The default is **x** (hexadecimal) initially. The default changes each time you use either **x** or **print**.

u The unit size is any of:

- **b** — bytes.
- **h** — halfwords (two bytes).
- **w** — words (four bytes). This is the initial default.
- **g** — giant words (eight bytes).

Each time you specify a unit size with **x**, that size becomes the default unit the next time you use **x**. (For the **s** and **i** formats, the unit size is ignored and isn’t normally written.)

addr The address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it’s always interpreted as an integer address of a byte of memory. See “Expressions” for more information on expressions. The default for *addr* is usually just after the last address examined—but several other commands also set the default address: **info breakpoints** (to the address of the last breakpoint listed), **info line** (to the starting address of a line), and **print** (if you use it to display a value from memory).

For example, **x/3uh 0x54320** is a request to display three halfwords (**h**) of memory, formatted as unsigned decimal integers (**u**), starting at address **0x54320**. The **x/4xw $sp** command prints the four words (**w**) of memory above the stack pointer (here, **$sp**; see "Registers") in hexadecimal (**x**).

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you don't have to remember whether unit size or format comes first; either order works. The output specifications **4xw** and **4wx** mean exactly the same thing. (However, the count *n* must come first; **wx4** doesn't work.)

Even though the unit size *u* is ignored for the formats **s** and **i**, you might still want to use a count *n*; for example, **3i** specifies that you want to see three machine instructions, including any operands. The command **disassemble** gives an alternative way of inspecting machine instructions; see "Source and machine code."

All the defaults for the arguments to **x** are designed to make it easy to continue scanning memory with minimal specifications each time you use **x**. For example, after you've inspected three machine instructions with **x/3i** *addr*, you can inspect the next seven with just **x/7**. If you use Enter to repeat the **x** command, the repeat count *n* is used again; the other arguments default as for successive uses of **x**.

The addresses and contents printed by the **x** command aren't saved in the value history because there's often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables **$_** and **$__**. After an **x** command, the last address examined is available for use in expressions in the convenience variable **$_**. The contents of that address, as examined, are available in the convenience variable **$__**.

If the **x** command has a repeat count, the address and contents saved are from the last memory unit printed; this isn't the same as the last address printed if several units were printed on the last line of output.

Automatic display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

This display shows item numbers, expressions and their current values. As with displays you request manually using **x** or **print**, you can specify the output format you prefer; in fact, **display** decides whether to use **print** or **x** depending on how elaborate your format specification is—it uses **x** if you specify a unit size, or one of the two formats (**i** and **s**) that are supported only by **x**; otherwise it uses **print**.

display *exp*
: Add the expression *exp* to the list of expressions to display each time your program stops. See "Expressions." The **display** command doesn't repeat if you press Enter again after using it.

display/*fmt exp*
: For *fmt* specifying only a display format and not a size or count, add the expression *exp* to the auto-display list but arrange to display it each time in the specified format *fmt*. See "Output formats."

display/*fmt addr*
: For *fmt* **i** or **s**, or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing **x/***fmt addr*. See "Examining memory."

For example, **display/i $pc** can be helpful, to see the machine instruction about to be executed each time execution stops (**$pc** is a common name for the program counter; see "Registers").

undisplay *dnums...*
delete display *dnums...*
: Remove item numbers *dnums* from the list of expressions to display.

  The **undisplay** command doesn't repeat if you press Enter after using it. (Otherwise you'd just get the error **No display number ...**.)

disable display *dnums...*
: Disable the display of item numbers *dnums*. A disabled display item isn't printed automatically, but isn't forgotten; it may be enabled again later.

enable display *dnums...*
: Enable the display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise.

display
: Display the current values of the expressions on the list, just as is done when your program stops.

info display
: Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions that wouldn't be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, it doesn't make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables isn't defined.

For example, if you give the command **display last_char** while inside a function with an argument *last_char*, GDB displays this argument while your program continues to stop inside that function. When it stops where there's no variable *last_char*, the display is disabled automatically. The next time your program stops where *last_char* is meaningful, you can enable the display expression once again.

Print settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

set print address
set print address on

GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is **on**. For example, this is what a stack frame display looks like with **set print address on**:

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote != def_lquote)
```

set print address off

Don't print addresses when displaying their contents. For example, this is the same stack frame displayed with **set print address off**:

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530         if (lquote != def_lquote)
```

You can use **set print address off** to eliminate all machine-dependent displays from the GDB interface. For example, with **print address off**, you should get the same text for backtraces on all machines—whether or not they involve pointer arguments.

show print address

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset. If that symbol doesn't uniquely identify the address (for example, it's a name whose scope is a single source file), you may need to clarify. One way to do this is with **info line**, for example **info line *0x4537**. Alternately, you can set GDB to print the source file and line number when it prints a symbolic address:

set print symbol-filename on

Tell GDB to print the source filename and line number of a symbol in the symbolic form of an address.

set print symbol-filename off

Don't print source filename and line number of a symbol. This is the default.

show print symbol-filename

Show whether or not GDB prints the source filename and line number of a symbol in the symbolic form of an address.

Another situation where it's helpful to show symbol filenames and line numbers is when disassembling code; GDB shows you the line number and source file that correspond to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

set print max-symbolic-offset *max-offset*

Tell GDB to display the symbolic form of an address only if the offset between the closest earlier symbol and the address is less than *max-offset*. The default is 0, which tells GDB to always print the symbolic form of an address if any symbol precedes it.

show print max-symbolic-offset

Ask how large the maximum offset is that GDB prints in a symbolic address.

If you have a pointer and you aren't sure where it points, try **set print symbol-filename on**. Then you can determine the name and source file location of the variable where it points, using **p/a** *pointer*. This interprets the address in symbolic form. For example, here GDB shows that a variable *ptt* points at another variable *t*, defined in **hi2.c**:

```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

For pointers that point to a local variable, **p/a** doesn't show the symbol name and filename of the referent, even with the appropriate **set print** options turned on.

Other settings control how different kinds of objects are printed:

set print array
set print array on

Pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

set print array off

Return to compressed format for arrays.

show print array

Show whether compressed or pretty format is selected for displaying arrays.

set print elements *number-of-elements*

Set a limit on how many elements of an array GDB prints. If GDB is printing a large array, it stops printing after it has printed the number of elements set by the **set print elements** command. This limit also applies to the display of strings. Setting *number-of-elements* to zero means that the printing is unlimited.

show print elements

Display the number of elements of a large array that GDB prints. If the number is 0, the printing is unlimited.

set print null-stop

Cause GDB to stop printing the characters of an array when the first NULL is encountered. This is useful when large arrays actually contain only short strings.

set print pretty on

Cause GDB to print structures in an indented format with one member per line, like this:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

set print pretty off

Cause GDB to print structures in a compact format, like this:

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, \
meat = 0x54 "Pork"}
```

This is the default format.

show print pretty

Show which format GDB is using to print structures.

set print sevenbit-strings on

Print using only seven-bit characters; if this option is set, GDB displays any eight-bit characters (in strings or character values) using the notation *nnn*. This setting is best if you're working in English (ASCII) and you use the high-order bit of characters as a marker or "meta" bit.

set print sevenbit-strings off

Print full eight-bit characters. This lets you use more international character sets, and is the default.

show print sevenbit-strings

Show whether or not GDB is printing only seven-bit characters.

set print union on

Tell GDB to print unions that are contained in structures. This is the default setting.

set print union off

Tell GDB not to print unions that are contained in structures.

show print union

Ask GDB whether or not it prints unions that are contained in structures. For example, given the declarations:

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
              Bug_forms;

struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};

struct thing foo = {Tree, {Acorn}};
```

with **set print union on** in effect, **p foo** prints:

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with **set print union off** in effect, it prints:

```
$1 = {it = Tree, form = {...}}
```

These settings are of interest when debugging C++ programs:

set print demangle
set print demangle on

Print C++ names in their source form rather than in the encoded ("mangled") form passed to the assembler and linker for type-safe linkage. The default is **on**.

show print demangle

Show whether C++ names are printed in mangled or demangled form.

set print asm-demangle
set print asm-demangle on

Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

show print asm-demangle

Show whether C++ names in assembly listings are printed in mangled or demangled form.

set demangle-style *style*

Choose among several encoding schemes used by different compilers to represent C++ names. The choices for *style* are:

- **auto** Allow GDB to choose a decoding style by inspecting your program.
- **gnu** Decode based on the GNU C++ compiler (**g++**) encoding algorithm. This is the default.
- **lucid** Decode based on the Lucid C++ compiler (**lcc**) encoding algorithm.
- **arm** Decode using the algorithm in the *C++ Annotated Reference Manual*.

  This setting alone isn't sufficient to allow debugging **cfront**-generated executables. GDB would require further enhancement to permit that.
- **foo** Show the list of formats.

show demangle-style

Display the encoding style currently in use for decoding C++ symbols.

set print object
set print object on

When displaying a pointer to an object, identify the *actual* (derived) type of the object rather than the *declared* type, using the virtual function table.

set print object off

Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

show print object

Show whether actual, or declared, object types are displayed.

set print static-members
set print static-members on

Print static members when displaying a C++ object. The default is on.

set print static-members off

Don't print static members when displaying a C++ object.

show print static-members

Show whether C++ static members are printed, or not.

set print vtbl
set print vtbl on

Pretty print C++ virtual function tables. The default is off.

set print vtbl off

Don't pretty print C++ virtual function tables.

show print vtbl

Show whether C++ virtual function tables are pretty printed, or not.

Value history

Values printed by the **print** command are saved in the GDB *value history*. This lets you refer to them in other expressions. Values are kept until the symbol table is reread or discarded (for example with the **file** or **symbol-file** commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given *history numbers*, which you can use to refer to them. These are successive integers starting with 1. The **print** command shows you the history number assigned to a value by printing **$***num* **=** before the value; here *num* is the history number.

To refer to any previous value, use **$** followed by the value's history number. The way **print** labels its output is designed to remind you of this. Just **$** refers to the most recent value in the history, and **$$** refers to the value before that. **$$***n* refers to the *n*th value from the end; **$$2** is the value just prior to **$$**, **$$1** is equivalent to **$$**, and **$$0** is equivalent to **$**.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type:

```
p *$
```

If you have a chain of structures where the component **next** points to the next one, you can print the contents of the next one with this:

```
p *$.next
```

You can print successive links in the chain by repeating this command—which you can do by just typing Enter.

The history records values, not expressions. If the value of **x** is 4 and you type these commands:

```
print x
set x=5
```

then the value recorded in the value history by the **print** command remains 4 even though the value of **x** has changed.

show values Print the last ten values in the value history, with their item numbers. This is like **p $$9** repeated ten times, except that **show values** doesn't change the history.

show values *n* Print ten history values centered on history item number *n*.

show values + Print ten history values just after the values last printed. If no more values are available, **show values +** produces no display.

Pressing Enter to repeat **show values** *n* has exactly the same effect as **show values +**.

Convenience variables

GDB provides *convenience variables* that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they aren't part of your program, and setting a convenience variable has no direct effect on further execution of your program. That's why you can use them freely.

Convenience variables are prefixed with **$**. Any name preceded by **$** can be used for a convenience variable, unless it's one of the predefined machine-specific register names (see "Registers"). Value history references, in contrast, are *numbers* preceded by **$**. See "Value history."

You can save a value in a convenience variable with an assignment expression, just as you'd set a variable in your program. For example:

```
set $foo = *object_ptr
```

saves in $*foo* the value contained in the object pointed to by *object_ptr*.

Using a convenience variable for the first time creates it, but its value is **void** until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign to a convenience variable any type of value, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

show convenience

Print a list of convenience variables used so far, and their values. Abbreviated **show con**.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

```
set $i = 0
print bar[$i++]->contents
```

Repeat that command by pressing Enter.

Some convenience variables are created automatically by GDB and given values likely to be useful:

$_
: The variable **$_** is automatically set by the **x** command to the last address examined (see "Examining memory"). Other commands that provide a default address for **x** to examine also set **$_** to that address; these commands include **info line** and **info breakpoint**. The type of **$_** is **void *** except when set by the **x** command, in which case it's a pointer to the type of **$__**.

$__
: The variable **$__** is automatically set by the **x** command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

$_exitcode
: The variable **$_exitcode** is automatically set to the exit code when the program being debugged terminates.

Registers

You can refer to machine register contents, in expressions, as variables with names starting with **$**. The names of registers are different for each machine; use **info registers** to see the names used on your machine.

info registers

Print the names and values of all registers except floating-point registers (in the selected stack frame).

info all-registers

Print the names and values of all registers, including floating-point registers.

info registers *regname* ...

Print the value of each specified register *regname*. As discussed in detail below, register values are normally relative to the selected stack frame. The *regname* may be any register name valid on the machine you're using, with or without the initial **$**.

GDB has four "standard" register names that are available (in expressions) on most machines—whenever they don't conflict with an architecture's canonical mnemonics for registers:

`$pc` Program counter.

`$sp` Stack pointer.

`$fp` A register that contains a pointer to the current stack frame.

`$ps` A register that contains the processor status.

For example, you could print the program counter in hex with:

```
p/x $pc
```

or print the instruction to be executed next with:

```
x/i $pc
```

or add four to the stack pointer with:

```
set $sp += 4
```

This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting **`$sp`** isn't allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use the Enter key.

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there's no conflict. The **`info registers`** command shows the canonical names.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers that can hold nothing but floating point; these registers are considered to have floating point values. There's no way to refer to the contents of an ordinary register as floating point value (although you can *print* it as a floating point value with **`print/f $`***regname*).

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system isn't the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" (raw) format, but all C programs expect to work with "double" (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the **`info registers`** command prints the data in both formats.

Normally, register values are relative to the selected stack frame (see "Selecting a frame"). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with **`frame 0`**).

However, GDB must deduce where registers are saved, from the machine code generated by your compiler. If some registers aren't saved, or if GDB is unable to locate the saved registers, the selected stack frame makes no difference.

Floating point hardware

Depending on the configuration, GDB may be able to give you more information about the status of the floating point hardware.

`info float`
: Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip. Currently, **`info float`** is supported on x86 machines.

Examining the symbol table

The commands described in this section allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and doesn't change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB (see the description of the **`gdb`** utility).

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see "Program variables"). Filenames are recorded in object files as debugging symbols, but GDB ordinarily parses a typical filename, like **`foo.c`**, as the three words **`foo`**, **`.`**, and **`c`**. To allow GDB to recognize **`foo.c`** as a single symbol, enclose it in single quotes. For example:

```
p 'foo.c'::x
```

looks up the value of *x* in the scope of the file **`foo.c`**.

`info address` *symbol*
: Describe where the data for *symbol* is stored. For a register variable, this says which register it's kept in. For a nonregister local variable, this prints the stack-frame offset at which the variable is always stored.

: Note the contrast with **`print &`***symbol*, which doesn't work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

`whatis` *exp*
: Print the data type of expression *exp*. The *exp* expression isn't actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it don't take place. See "Expressions."

whatis
: Print the data type of **$**, the last value in the value history.

ptype *typename*
: Print a description of data type *typename*, which may be the name of a type, or for C code it may have the form:

- **class** *class-name*
- **struct** *struct-tag*
- **union** *union-tag*
- **enum** *enum-tag*

ptype *exp*
ptype
: Print a description of the type of expression *exp*. The **ptype** command differs from **whatis** by printing a detailed description, instead of just the name of the type. For example, for this variable declaration:

```
struct complex {double real; double imag;} v;
```

the two commands give this output:

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with **whatis**, using **ptype** without an argument refers to the type of **$**, the last value in the value history.

info types *regexp*
info types
: Print a brief description of all types whose name matches *regexp* (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, **i type value** gives information on all types in your program whose name includes the string **value**, but **i type ^value$** gives information only on types whose complete name is **value**.

This command differs from **ptype** in two ways: first, like **whatis**, it doesn't print a detailed description; second, it lists all source files where a type is defined.

info source
: Show the name of the current source file—that is, the source file for the function containing the current point of execution—and the language it was written in.

info sources
: Print the names of all source files in your program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols are read when needed.

info functions Print the names and data types of all defined functions.

info functions *regexp*

Print the names and data types of all defined functions whose names contain a match for regular expression *regexp*. Thus, **info fun step** finds all functions whose names include **step**; **info fun ^step** finds those whose names start with **step**.

info variables Print the names and data types of all variables that are declared outside of functions (i.e. excluding local variables).

info variables *regexp*

Print the names and data types of all variables (except for local variables) whose names contain a match for regular expression *regexp*.

Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. If you're running on one of these systems, you can allow GDB to reload the symbols for automatically relinked modules:

- **set symbol-reloading on** — replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.
- **set symbol-reloading off** — don't replace symbol definitions when reencountering object files of the same name. This is the default state; if you aren't running on a system that permits automatically relinking modules, you should leave **symbol-reloading** off, since otherwise GDB may discard symbols when linking large programs, that may contain several modules (from different directories or libraries) with the same name.
- **show symbol-reloading** — show the current **on** or **off** setting.

maint print symbols *filename*
maint print psymbols *filename*
maint print msymbols *filename*

Write a dump of debugging symbol data into the file *filename*. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included.

- If you use **maint print symbols**, GDB includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols GDB has read. You can use the command **info sources** to find out which files these are.

- If you use **`maint print psymbols`** instead, the dump shows information about symbols that GDB only knows partially—that is, symbols defined in files that GDB has skimmed, but not yet read completely.
- Finally, **`maint print msymbols`** dumps just the minimal symbol information required for each object file from which GDB has read some symbols.

Altering execution

Once you think you've found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experimenting, using the GDB features for altering execution of the program.

For example, you can store new values in variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function.

Assignment to variables

To alter the value of a variable, evaluate an assignment expression. See "Expressions". For example,

```
print x=4
```

stores the value 4 in the variable *x* and then prints the value of the assignment expression (which is 4).

If you aren't interested in seeing the value of the assignment, use the **`set`** command instead of the **`print`** command. The **`set`** command is really the same as **`print`** except that the expression's value isn't printed and isn't put in the value history (see "Value history"). The expression is evaluated only for its effects.

If the beginning of the argument string of the **`set`** command appears identical to a **`set`** subcommand, use the **`set variable`** command instead of just **`set`**. This command is identical to **`set`** except for its lack of subcommands. For example, if your program has a variable *width*, you get an error if you try to set a new value with just **`set width=13`**, because GDB has the command **`set width`**:

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

The invalid expression, of course, is **`=47`**. In order to actually set the program's variable *width*, use:

```
(gdb) set var width=47
```

GDB allows more implicit conversions in assignments than C; you can freely store an integer value into a pointer variable or vice versa, and you can convert any structure to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the **`{...}`** construct to generate a value of specified type at a specified address (see "Expressions"). For example, **`{int}0x83040`** refers to memory location **`0x83040`** as an integer (which implies a certain size and representation in memory), and:

```
set {int}0x83040 = 4
```

stores the value 4 in that memory location.

Continuing at a different address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the **`continue`** command. You can instead continue at an address of your own choosing, with the following commands:

`jump` *linespec*
: Resume execution at line *linespec*. Execution stops again immediately if there's a breakpoint there. See "Printing source lines" for a description of the different forms of *linespec*.

  The **`jump`** command doesn't change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line *linespec* is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the **`jump`** command requests confirmation if the specified line isn't in the function currently executing. However, even bizarre results are predictable if you're well acquainted with the machine-language code of your program.

`jump *`*address*
: Resume execution at the instruction at *address*.

You can get much the same effect as the **`jump`** command by storing a new value in the register **`$pc`**. The difference is that this doesn't start your program running; it only changes the address of where it *will* run when you continue. For example:

```
set $pc = 0x485
```

makes the next **`continue`** command or stepping command execute at address **`0x485`**, rather than at the address where your program stopped. See "Continuing and stepping."

The most common occasion to use the **`jump`** command is to back up — perhaps with more breakpoints set — over a portion of a program that has already executed, in order to examine its execution in more detail.

Giving your program a signal

`signal` *signal*
: Resume execution where your program stopped, but immediately give it the given *signal*. The *signal* can be the name or number of a signal. For example, on many systems **`signal 2`** and **`signal SIGINT`** are both ways of sending an interrupt signal.

  Alternatively, if *signal* is zero, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinary see the signal when resumed with the **`continue`** command; **`signal 0`** causes it to resume without a signal.

  The **`signal`** command doesn't repeat when you press Enter a second time after executing the command.

Invoking the **`signal`** command isn't the same as invoking the **`kill`** utility from the shell. Sending a signal with **`kill`** causes GDB to decide what to do with the signal depending on the signal handling tables (see "Signals"). The **`signal`** command passes the signal directly to your program.

Returning from a function

`return`
`return` *expression*

You can cancel the execution of a function call with the **`return`** command. If you give an *expression* argument, its value is used as the function's return value.

When you use **`return`**, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to **`return`**.

This pops the selected stack frame (see "Selecting a frame") and any other frames inside it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The **`return`** command doesn't resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the **`finish`** command (see "Continuing and stepping") resumes execution until the selected stack frame returns naturally.

Calling program functions

`call` *expr*
: Evaluate the expression *expr* without displaying **`void`** returned values.

You can use this variant of the **`print`** command if you want to execute a function from your program, but without cluttering the output with **`void`** returned values. If the result isn't void, it's printed and saved in the value history.

A user-controlled variable, *call_scratch_address*, specifies the location of a scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack doesn't work in systems that have separate instruction and data spaces.

Patching programs

By default, GDB opens the file containing your program's executable code (or the core file) read-only. This prevents accidental alterations to machine code; but it also prevents you from intentionally patching your program's binary.

If you'd like to be able to patch the binary, you can specify that explicitly with the **`set write`** command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

`set write on`
`set write off`
: If you specify **`set write on`**, GDB opens executable and core files for both reading and writing; if you specify **`set write off`** (the default), GDB opens them read-only.

  If you've already loaded a file, you must load it again (using the **`exec-file`** or **`core-file`** command) after changing **`set write`** for your new setting to take effect.

`show write`
: Display whether executable files and core files are opened for writing as well as reading.

Appendix B

ARM Memory Management

In this appendix. . .

This appendix describes various features and restrictions related to the Neutrino implementation on ARM/Xscale processors:

- restrictions and issues that don't apply to other processor ports, and may need to be taken into consideration when porting code to ARM/Xscale targets.
- ARM-specific features that you can use to work around some of the restrictions imposed by the Neutrino ARM implementation

For an overview of how Neutrino manages memory, see the introduction to the Finding Memory Errors chapter of the IDE *User's Guide*.

ARM-specific restrictions and issues

This section describes the major restrictions and issues raised by the Neutrino implementation on ARM/Xscale:

- behavior of _NTO_TCTL_IO
- implications of the ARM/Xscale cache architecture

_NTO_TCTL_IO behavior

Device drivers in Neutrino use *ThreadCtl()* with the _NTO_TCTL_IO flag to obtain I/O privileges. This mechanism allows direct access to I/O ports and the ability to control processor interrupt masking.

On ARM platforms, all I/O access is memory-mapped, so this flag is used primarily to allow manipulation of the processor interrupt mask.

Normal user processes execute in the processor's User mode, and the processor silently ignores any attempts to manipulate the interrupt mask in the CPSR register (i.e. they don't cause any protection violation, and simply have no effect on the mask).

The _NTO_TCTL_IO flag makes the calling thread execute in the processor's System mode. This is a privileged mode that differs only from the Supervisor mode in its use of banked registers.

This means that such privileged user processes execute with all the access permission of kernel code:

- They can directly access kernel memory:
  - They fault if they attempt to write to read-only memory.
  - They don't fault if they write to writable mappings. This includes kernel data and also the mappings for page tables.
- They can circumvent the regular permission control for user mappings:
  - They don't fault if they write to read-only user memory.

The major consequence of this is that buggy programs using _NTO_TCTL_IO can corrupt kernel memory.

Implications of the ARM Cache Architecture

All currently supported ARM/Xscale processors implement a virtually indexed cache. This has a number of software-visible consequences:

- Whenever any virtual-to-physical address translations are changed, the cache must be flushed, because the contents of the cache no longer identify the same physical memory. This would typically have to be performed:
  - when memory is unmapped (to prevent stale cache data)
  - during a context switch (since all translations have now changed).

  The Neutrino implementation does perform this flushing when memory is unmapped, but it avoids the context-switch penalty by using the "Fast Context Switch Extension" implemented by some ARM MMUs. This is described below.

- Shared memory accessed via different virtual addresses may need to be made uncached, because the cache would contain different entries for each virtual address range. If any of these mappings are writable, it causes a coherency problem because modifications made through one mapping aren't visible through the cache entries for other mappings.

- Memory accessed by external bus masters (e.g. DMA) may need to be made uncached:
  - If the DMA writes to memory, it will be more up to date than a cache entry that maps that memory. CPU access would get stale data from the cache.
  - If the DMA reads from memory, it may be stale if there is a cache entry that maps that memory. DMA access would get stale data from memory.

  An alternative to making such memory uncached is to modify all drivers that perform DMA access to explicitly synchronize memory when necessary:

  - before a DMA read from memory: clean and invalidate cache entries
  - after a DMA write to memory: invalidate cache entries

As mentioned, Neutrino uses the MMU Fast Context Switch Extension (FCSE) to avoid cache-flushing during context switches. Since the cost of this cache-flushing can be significant (potentially many thousands of cycles), this is crucial to a microkernel system like Neutrino because context switches are much more frequent than in a monolithic (e.g. UNIX-like) OS:

- Message passing involves context switching between sender and receiver.
- Interrupt handling involves context switching to the driver address space.

The FCSE implementation works by splitting the 4 GB virtual address space into a number of 32 MB slots. Each address space appears to have a virtual address space range of 0 - 32 MB, but the MMU transparently remaps this to a a "real" virtual address by putting the slot index into the top 7 bits of the virtual address.

For example, consider two processes: process 1 has slot index 1; process 2 has slot index 2. Each process appears to have an address space 0 - 32 MB, and their code uses those addresses for execution, loads and stores.
In reality, the virtual addresses seen by the MMU (cache and TLB) are:

- Process 1: `0x00000000-0x01FFFFFF` is mapped to `0x02000000-0x03FFFFFF`.
- Process2: `0x00000000-0x01FFFFFF` is mapped to `0x04000000-0x07FFFFFF`.

This mechanism imposes a number of restrictions:

- Each process address space is limited to 32 MB in size. This space contains all the code, data, heap, thread stacks and shared objects mapped by the process. The virtual address space is allocated as follows:

Range:	Used for:
0– 1 MB	Initial thread stack
1–16 MB	Program text, data, and BSS
16–24 MB	Shared libraries
24–32 MB	MAP_SHARED mappings

  When a program is loaded, the loader will have populated the stack, text, data, BSS, and shared library areas.

  If you allocate memory, *malloc()* tries to find a free virtual address range for the requested size. If you try to allocate more than 15 MB, the allocation will likely fail because of this layout. The free areas are typically:

  - approximately 15 MB (addresses between 1 MB + **`sizeof(text/data/heap)`** and the 16 MB boundary)
  - approximately 7 MB (addresses between 16 MB + **`sizeof(shared libs)`** and the 24 MB boundary)
  - approximately 8 MB (addresses between 24 MB + **`sizeof(MAP_SHARED mappings)`** and the 32 MB boundary)

- The FCSE remapping uses the top 7 bits of the address space, which means there can be at most 128 slots. In practice, some of the 4 GB virtual space is required for the kernel, so the real number is lower.

  The current limit is 63 slots:

  - Slot 0 is never used.
  - Slots 64-127 (`0x80000000-0xFFFFFFFF`) are used by the kernel and the ARM-specific *shm_ctl()* support described below.

  Since each process typically has its own address space, this imposes a hard limit of at most 63 different processes.

- Because the MMU transparently remaps each process's virtual address, shared memory objects must be mapped uncached, since they're always mapped at different virtual addresses.

  Strictly speaking, this is required only if at least one writable mapping exists, but the current VM implementation doesn't track this, and unconditionally makes all mappings uncached.

  The consequence of this is that performance of memory accesses to shared memory object mappings will be bound by the uncached memory performance of the system.

ARM-specific features

This section describes the ARM-specific behavior of certain operations that are provided via a processor-independent interface:

- *shm_ctl()* operations for defining special memory object properties

shm_ctl() behavior

The Neutrino implementation on ARM uses various *shm_ctl()* flags to provide some workarounds for the restrictions imposed by the MMU FCSE implementation, to provide a "global" address space above `0x80000000` that lets processes map objects that wouldn't otherwise fit into the (private) 32 MB process-address space.

The following flags supplied to *shm_ctl()* create a shared memory object that you can subsequently *mmap()* with special properties:

- You can use SHMCTL_PHYS to create an object that maps a physical address range that's greater than 32 MB. A process that maps such an object gets a (unique) mapping of the object in the "global address space."

- You can use SHMCTL_GLOBAL to create an object whose "global address space" mapping is the same for all processes. This address is allocated when the object is first mapped, and subsequent maps receive the virtual address allocated by the first mapping.

  Since all mappings of these objects share the same virtual address, there are a number of artifacts caused by *mmap()*:

  - If PROT_WRITE is specified, the mappings are made writable. This means all processes that have mapped now have writable access even if they initially mapped it PROT_READ only.
  - If PROT_READ only is specified, the mappings aren't changed. If this is the first *mmap()*, the mappings are made read-only, otherwise the mappings are unchanged.
  - If PROT_NOCACHE isn't specified, the mappings are allowed to be cacheable since all processes share the same virtual address, and hence no cache aliases will exist.

- SHMCTL_LOWERPROT causes a *mmap()* of the object to have user-accessible mappings. By default, system-level mappings are created, which allow access only by threads that used _NTO_TCTL_IO.

  Specifying this flag allows *any* process in the system to access the object, because the virtual address is visible to all processes.

To create these special mappings:

1 Create and initialize the object:

```
fd = shm_open(name, ...)
shm_ctl(fd, ...)
```

Note that you must be **root** to use *shm_ctl()*.

2 Map the object:

```
fd = shm_open(name, ...)
mmap( ..., fd, ...)
```

Any process that can use *shm_open()* on the object can map it, not just the process that created the object.

The following table summarizes the effect of the various combinations of flags passed to *shm_ctl()*:

Flags	**Object type**	**Effect of *mmap()***
SHMCTL_ANON	Anonymous memory (not contiguous)	Mapped into normal process address space. PROT_NOCACHE is forced.
SHMCTL_ANON | SHMCTL_PHYS	Anonymous memory (physically contiguous)	Mapped into normal process address space. PROT_NOCACHE is forced.
SHMCTL_ANON | SHMCTL_GLOBAL	Anonymous memory (not contiguous)	Mapped into global address space. PROT_NOCACHE isn't forced. All processes receive the same mapping.
SHMCTL_ANON | SHMCTL_GLOBAL | SHMCTL_PHYS	Anonymous memory (not contiguous)	Mapped into global address space. PROT_NOCACHE isn't forced. All processes receive the same mapping.
SHMCTL_PHYS	Physical memory range	Mapped into global address space. PROT_NOCACHE is forced. Processes receive unique mappings.

continued...

Flags	**Object type**	**Effect of *mmap()***
`SHMCTL_PHYS | SHMCTL_GLOBAL`	Physical memory range	Mapped into global address space. PROT_NOCACHE isn't forced. All processes receive the same mapping.

Note that by default, *mmap()* creates privileged access mappings, so the caller must have _NTO_TCTL_IO privilege to access them.

Flags may specify SHMCTL_LOWERPROT to create user-accessible mappings. However, this allows any process to access these mappings if they're in the global address space.

Appendix C

Advanced Qnet Topics

In this appendix...

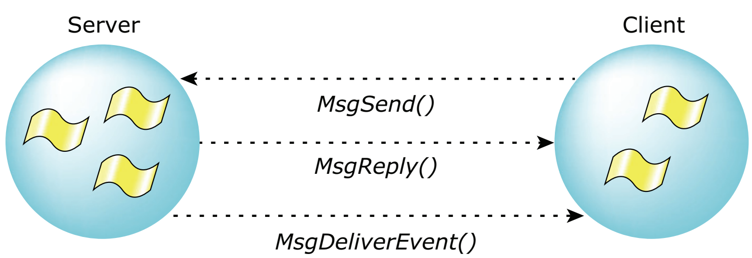
Low-level discussion on Qnet principles

The Qnet protocol extends interprocess communication (IPC) transparently over a network of microkernels. This is done by taking advantage of the Neutrino's message-passing paradigm. Message passing is the central theme of Neutrino that manages a group of cooperating processes by routing messages. This enhances the efficiency of all transactions among all processes throughout the system.

As we found out in the "How does it work?" section of the Transparent Distributed Processing Using Qnet chapter, many POSIX and other function calls are built on this message passing. For example, the *write()* function is built on the *MsgSendv()* function. In this section, you'll find several things, e.g. how Qnet works at the message passing level; how node names are resolved to node numbers, and how that number is used to create a connection to a remote node.

In order to understand how message passing works, consider two processes that wish to communicate with each other: a client process and a server process. First we consider a single-node case, where both client and server reside in the same machine. In this case, the client simply creates a connection (via *ConnectAttach()*) to the server, and then sends a message (perhaps via *MsgSend()*).

The Qnet protocol extends this message passing over to a network. For example, consider the case of a simple network with two machines: one contains the client process, the other contains the server process. The code required for client-server communication is identical (it uses same API) to the code in the single-node case. The client creates a connection to the server and sends the server a message. The only difference in the network case is that the client specifies a different node descriptor for the *ConnectAttach()* function call in order to indicate the server's node. See the diagram below to understand how message passing works.



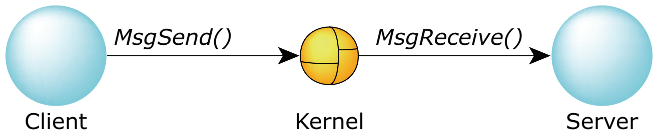
Each node in the network is assigned a unique name that becomes its identifier. This is what we call a *node descriptor*. This name is the only visible means to determine whether the OS is running as a network or as a standalone operating system.

Details of Qnet data communication

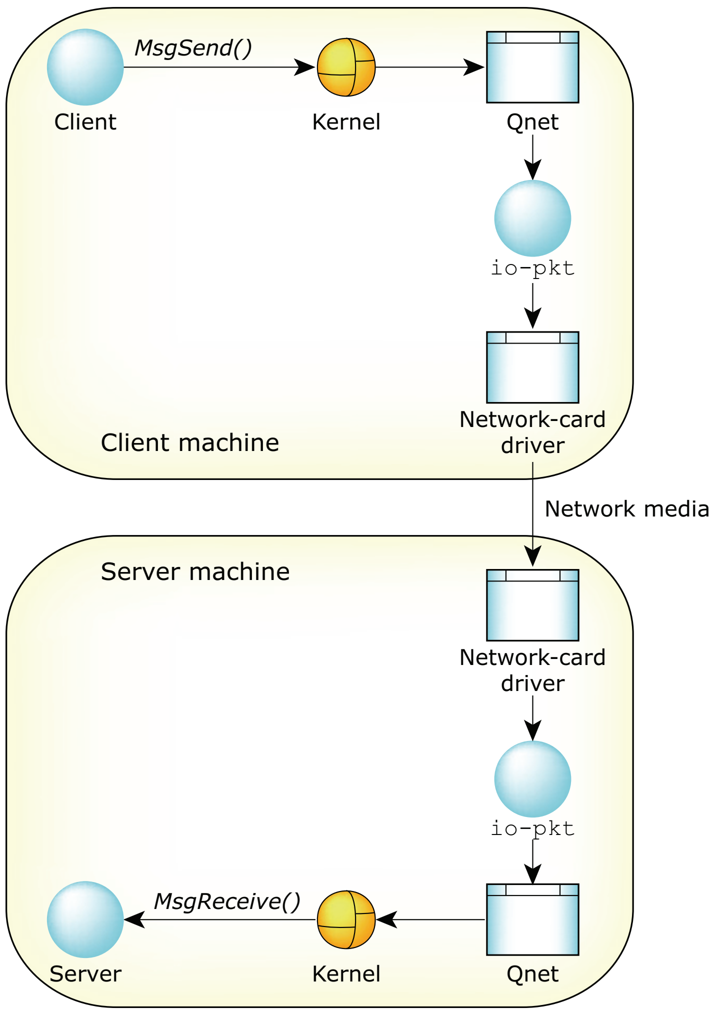
As mentioned before, Qnet relies on the message passing paradigm of Neutrino. Before any message pass, however, the application (e.g. the client) must establish a connection to the server using the low-level *ConnectAttach()* function call:

```
ConnectAttach(nd, pid, chid, index, flags);
```

In the above call, *nd* is the node descriptor that identifies each node uniquely. The node descriptor is the only visible means to determine whether the Neutrino is running as a network or as a standalone operating system. If *nd* is zero, you're specifying a local server process, and you'll get local message passing from the client to the server, carried out by the local kernel as shown below:



When you specify a nonzero value for *nd*, the application transparently passes message to a server on another machine, and connects to a server on another machine. This way, Qnet not only builds a network of trusted machines, it lets all these machines share their resources with little overhead.



The advantage of this approach lies in using the same API. The key design features are:

- The kernel puts the user data directly into (and out of) the network card's buffers - there's no copying of the payload.
- There are no context switches as the packet travels from (and to) the kernel from the network card.

These features maximize performance for large payloads and minimize turnaround time for small packets.

Node descriptors

The <sys/netmgr.h> header file

The **<sys/netmgr.h>** header defines the ND_LOCAL_NODE macro as zero. You can use it any time that you're dealing with node descriptors to make it obvious that you're talking about the local node.

As discussed, node descriptors represent machines, but they also include *Quality of Service* information. If you want to see if two node descriptors refer to the same machine, you can't just arithmetically compare the descriptors for equality; use the *ND_NODE_CMP()* macro instead:

- If the return value from the macro is zero, the descriptors refer to the same node.
- If the value is less than 0, the first node is "less than" the second.
- If the value is greater than 0, the first node is "greater than" the second.

This is similar to the way that *strcmp()* and *memcmp()* work. It's done this way in case you want to do any sorting that's based on node descriptors.

The **<sys/netmgr.h>** header file also defines the following networking functions:

- *netmgr_strtond()*
- *netmgr_ndtostr()*
- *netmgr_remote_nd()*

netmgr_strtond()

```
int netmgr_strtond(const char *nodename, char **endstr);
```

This function converts the string pointed at by *nodename* into a node descriptor, which it returns. If there's an error, *netmgr_strtond()* returns -1 and sets *errno*. If the *endstr* parameter is non-NULL, *netmgr_strtond()* sets **endstr* to point at the first character beyond the end of the node name. This function accepts all three forms of node name — simple, directory, and FQNN (Fully Qualified NodeName). FQNN identifies a Neutrino node using a unique name on a network. The FQNN consists of the nodename and the node domain.

netmgr_ndtostr()

```
int netmgr_ndtostr(unsigned flags,
                   int nd,
                   char *buf,
                   size_t maxbuf);
```

This function converts the given node descriptor into a string and stores it in the memory pointed to by *buf*. The size of the buffer is given by *maxbuf*. The function returns the actual length of the node name (even if the function had to truncate the name to get it to fit into the space specified by *maxbuf*), or -1 if an error occurs (*errno* is set).

The *flags* parameter controls the conversion process, indicating which pieces of the string are to be output. The following bits are defined:

ND2S_DIR_SHOW,
ND2S_DIR_HIDE
: Show or hide the network directory portion of the string. If you don't set either of these bits, the string includes the network directory portion if the node isn't in the default network directory.

ND2S_QOS_SHOW,
ND2S_QOS_HIDE
: Show or hide the quality of service portion of the string. If you don't specify either of these bits, the string includes the quality of service portion if it isn't the default QoS for the node.

ND2S_NAME_SHOW,
ND2S_NAME_HIDE
: Show or hide the node name portion of the string. If you don't specify either of these bits, the string includes the name if the node descriptor doesn't represent the local node.

ND2S_DOMAIN_SHOW,
ND2S_DOMAIN_HIDE
: Show or hide the node domain portion of the string. If you don't specify either of these bits, and a network directory portion is included in the string, the node domain is included if it isn't the default for the output network directory. If you don't specify either of these bits, and the network directory portion isn't included in the string, the node domain is included if the domain isn't in the default network directory.

By combining the above bits in various combinations, all sorts of interesting information can be extracted, for example:

ND2S_NAME_SHOW
: A name that's useful for display purposes.

ND2S_DIR_HIDE | ND2S_NAME_SHOW | ND2S_DOMAIN_SHOW
: A name that you can pass to another node and know that it's referring to the same machine (i.e. the FQNN).

ND2S_DIR_SHOW | ND2S_NAME_HIDE | ND2S_DOMAIN_HIDE with ND_LOCAL_NODE
: The default network directory.

ND2S_DIR_HIDE | NDS2_QOS_SHOW | ND2S_NAME_HIDE | ND2S_DOMAIN_HIDE with ND_LOCAL_NODE
: The default Quality of Service for the node.

netmgr_remote_nd()

```
int netmgr_remote_nd(int remote_nd, int local_nd);
```

This function takes the *local_nd* node descriptor (which is relative to this node) and returns a new node descriptor that refers to the same machine, but is valid only for the node identified by *remote_nd*. The function can return -1 in some cases (e.g. if the *remote_nd* machine can't talk to the *local_nd* machine).

Booting over the network

Overview

Unleash the power of Qnet to boot your computer (i.e. client) over the network! You can do it when your machine doesn't have a local disk or large flash. In order to do this, you first need the GRUB executable. GRUB is the generic boot loader that runs at computer startup and is responsible for loading the OS into memory and starting to execute it.

During booting, you need to load the GRUB executable into the memory of your machine, by using:

- a GRUB floppy or CD (i.e. local copy of GRUB)

  Or:

- Network card boot ROM (e.g. PXE, bootp downloads GRUB from server)

Neutrino doesn't ship GRUB. To get GRUB:

1. Go to **www.gnu.org/software/grub** website.
2. Download the GRUB executable.
3. Create a floppy or CD with GRUB on it, or put the GRUB binary on the server for downloading by a network boot ROM.

Here's what the PXE boot ROM does to download the OS image:

- The network card of your computer broadcasts a DHCP request.

- The DHCP server responds with the relevant information, such as IP address, netmask, location of the **pxegrub** server, and the menu file.
- The network card then sends a TFTP request to the **pxegrub** server to transfer the OS image to the client.

Here's an example to show the different steps to boot your client using PXE boot ROM:

Creating directory and setting up configuration files

Create a new directory on your DHCP server machine called **/tftpboot** and run **make install**. Copy the **pxegrub** executable image from **/opt/share/grub/i386-pc** to the **/tftpboot** directory.

Modify the **/etc/dhcpd.conf** file to allow the network machine to download the **pxegrub** image and configuration menu, as follows:

```
# dhcpd.conf
#
# Sample configuration file for PXE dhcpd
#

subnet 192.168.0.0 netmask 255.255.255.0 {
  range 192.168.0.2 192.168.0.250;
  option broadcast-address 192.168.0.255;
  option domain-name-servers 192.168.0.1;
}

# Hosts which require special configuration options can be listed in
# host statements.   If no address is specified, the address will be
# allocated dynamically (if possible), but the host-specific information
# will still come from the host declaration.

host testpxe {
  hardware ethernet 00:E0:29:88:0D:D3;         # MAC address of system to boot
  fixed-address 192.168.0.3;                   # This line is optional
  option option-150 "(nd)/tftpboot/menu.1st";  # Tell grub to use Menu file
  filename "/tftpboot/pxegrub";                # Location of PXE grub image
}
# End dhcpd.conf
```

If you're using an ISC 3 DHCP server, you may have to add a definition of code 150 at the top of the **dhcpd.conf** file as follows:

option pxe-menu code 150 = text;

Then instead of using **option option-150**, use:

option pxe-menu "(nd)/tftpboot/menu.1st";)

Here's an example of the **menu.1st** file:

```
# menu.1st start

default 0                         # default OS image
to load
timeout 3                         # seconds to pause
before loading default image
title Neutrino Bios image         # text displayed in menu
```

```
kernel (nd)/tftpboot/bios.ifs         # OS image
title Neutrino ftp image              # text for second OS image
kernel (nd)/tftpboot/ftp.ifs          # 2nd OS image (optional)

# menu.1st end
```

Building an OS image

In this section, there is a functional buildfile that you can use to create an OS image that can be loaded by GRUB without a hard disk or any local storage.

Create the image by typing the following:

```
$ mkifs -vvv build.txt build.img
$ cp build.img /tftpboot
```

Here is the buildfile:

In a real buildfile, you can't use a backslash (\) to break a long line into shorter pieces, but we've done that here, just to make the buildfile easier to read.

```
[virtual=x86,elf +compress] boot = {
    startup-bios

    PATH=/proc/boot:/bin:/usr/bin:/sbin:/usr/sbin: \
    /usr/local/bin:/usr/local/sbin \
    LD_LIBRARY_PATH=/proc/boot: \
    /lib:/usr/lib:/lib/dll  procnto
}

[+script] startup-script = {
    procmgr_symlink ../../proc/boot/libc.so.3 /usr/lib/ldqnx.so.2

    #
    # do magic required to set up PnP and pci bios on x86
    #
    display_msg Do the BIOS magic ...
    seedres
    pci-bios
    waitfor /dev/pci

    #
    # A really good idea is to set hostname and domain
    # before qnet is started
    #
    setconf _CS_HOSTNAME aboyd
    setconf _CS_DOMAIN   ott.qnx.com

    #
    # If you do not set the hostname to something
    # unique before qnet is started, qnet will try
    # to create and set the hostname to a hopefully
    # unique string constructed from the ethernet
    # address, which will look like EAc07f5e
    # which will probably work, but is pretty ugly.
    #

    #
    # start io-pkt, network driver and qnet
    #
    # NB to help debugging, add verbose=1 after -pqnet below
    #
    display_msg Starting io-pkt-v6-hc and speedo driver and qnet ...
    io-pkt-v6-hc -dspeedo -pqnet

    display_msg Waiting for ethernet driver to initialize ...
    waitfor /dev/io-pkt/en0 60
```

```
    display_msg Waiting for Qnet to initialize ...
    waitfor /net 60

    #
    # Now that we can fetch executables from the remote server
    # we can run devc-con and ksh, which we do not include in
    # the image, to keep the size down
    #
    # In our example, the server we are booting from
    # has the hostname qpkg and the SAME domain: ott.qnx.com
    #
    # We clean out any old bogus connections to the qpkg server
    # if we have recently rebooted quickly, by fetching a trivial
    # executable which works nicely as a sacrificial lamb
    #
    /net/qpkg/bin/true

    #
    # now print out some interesting techie-type information
    #
    display_msg hostname:
    getconf _CS_HOSTNAME
    display_msg domain:
    getconf _CS_DOMAIN
    display_msg uname -a:
    uname -a

    #
    # create some text consoles
    #
    display_msg .
    display_msg Starting 3 text consoles which you can flip
    display_msg between by holding ctrl alt + OR ctrl alt -
    display_msg .
    devc-con -n3
    waitfor /dev/con1

    #
    # start up some command line shells on the text consoles
    #
    reopen /dev/con1
    [+session] TERM=qansi HOME=/ PATH=/bin:/usr/bin:\
    /usr/local/bin:/sbin:/usr/sbin:/usr/local/sbin:\
    /proc/boot ksh &

    reopen /dev/con2
    [+session] TERM=qansi HOME=/ PATH=/bin:/usr/bin:\
    /usr/local/bin:/sbin:/usr/sbin:\
    /usr/local/sbin:/proc/boot ksh &

    reopen /dev/con3
    [+session] TERM=qansi HOME=/ PATH=/bin:\
    /usr/bin:/usr/local/bin:/sbin:/usr/sbin:\
    /usr/local/sbin:/proc/boot ksh &

    #
    # startup script ends here
    #
}

#
# Let's create some links in the virtual file system so that
# applications are fooled into thinking there's a local hard disk
#

#
# Make /tmp point to the shared memory area
#
[type=link] /tmp=/dev/shmem

#
# Redirect console (error) messages to con1
#
[type=link] /dev/console=/dev/con1
```

```
#
# Now for the diskless qnet magic.  In this example, we are booting
# using a server which has the hostname qpkg.  Since we do not have
# a hard disk, we will create links to point to the servers disk
#
[type=link] /bin=/net/qpkg/bin
[type=link] /boot=/net/qpkg/boot
[type=link] /etc=/net/qpkg/etc
[type=link] /home=/net/qpkg/home
[type=link] /lib=/net/qpkg/lib
[type=link] /opt=/net/qpkg/opt
[type=link] /pkgs=/net/qpkg/pkgs
[type=link] /root=/net/qpkg/root
[type=link] /sbin=/net/qpkg/sbin
[type=link] /usr=/net/qpkg/usr
[type=link] /var=/net/qpkg/var
[type=link] /x86=/

#
# These are essential shared libraries which must be in the
# image for us to start io-pkt, the ethernet driver and qnet
#
libc.so.2
libc.so
devn-speedo.so
lsm-qnet.so

#
# copy code and data for all following executables
# which will be located in /proc/boot in the image
#
[data=copy]

seedres
pci-bios
setconf
io-pkt-v6-hc
waitfor

# uncomment this for debugging
# getconf
```

Booting the client

With your DHCP server running, boot the client machine using the PXE ROM. The client machine attempts to obtain an IP address from the DHCP server and load **pxegrub**. If successful, it should display a menu of available images to load. Select your option for the OS image. If you don't select any available option, the BIOS image is loaded after 3 seconds. You can also use the arrow keys to select the downloaded OS image.

If all goes well, you should now be running your OS image.

Troubleshooting

If the boot is unsuccessful, troubleshoot as follows:

Make sure your:

- DHCP server is running and is configured correctly
- **TFTP** isn't commented out of the **/etc/inetd.conf** file
- all users can read **pxegrub** and the OS image

- **inetd** is running

What are the limitations ...

- Qnet's functionality is limited when applications create a shared-memory region. That only works when the applications run on the same machine.
- Server calls such as *MsgReply()*, *MsgError()*, *MsgWrite()*, *MsgRead()*, and *MsgDeliverEvent()* behave differently for local and network cases. In the local case, these calls are *non blocking*, whereas in the network case, these calls *block*. In the non blocking scenario, a lower priority thread won't run; in the network case, a lower priority thread can run.
- The **mq** isn't working.
- The *ConnectAttach()* function appears to succeed the first time, even if the remote node is nonoperational or is turned off. In this case, it *should* report a failure, but it doesn't. For efficiency, *ConnectAttach()* is paired up with *MsgSend()*, which in turn reports the error. For the first transmission, packets from both *ConnectAttach()* and *MsgSend()* are transmitted together.
- Qnet isn't appropriate for broadcast or multicast applications. Since you're sending messages on specific channels that target specific applications, you can't send messages to more than one node or manager at the same time.
- For cross-endian development:
  - Qnet has limited support for communication between a big-endian and a little-endian machine; however, it is supported between machines of different processor types (e.g. ARMLE, x86) that are of the same endian. If you require cross-endian networking with Qnet, you need to be aware of these limitations:
    - Not all QNX resource managers support cross-endian. The ones that support cross-endian are: **pipe**, **mqueue**, **HAM**, **io-char**, **devf**, ETFS, and parts of **proc** (name resolve in procnto, **/dev/shmem**, **pathmgr** and **spawning** handle cross-endian messages, but **procfs** doesn't.)
    - For servers that use only QNX messages, you'll need to set the cross-endian flag RESMGR_FLAG_CROSS_ENDIAN in the **resmgr_attr_t** structure that you pass to the function *resmgr_attach()* in order to identify it as a cross-endian capable server. The actual byte-swapping code is done in **libc**.

Only the servers need to have the cross-endian flag RESMGR_FLAG_CROSS_ENDIAN set; the clients don't require this flag to be set.

    - If a server uses custom messages (i.e. **devctls**), the server will need to be modified to handle different endian messages. Incoming messages will contain a flag to identify whether it is the "other" endian (the big or the little endian). The server would be responsible for doing the endian swap for proper consumption. The server is also responsible for replying in the correct endian of the client. The servers can access the endian swap code that is in **libc**.

- You'll need to make the **fs-flash3** library endian-aware.
- There is a requirement for *readdir()* processing in order for the server to handle requests. You'll need to issue one *resmgr_msgreplyv()* rather than using *MsgWrite()* one at a time.

Glossary

A20 gate

On x86-based systems, a hardware component that forces the A20 address line on the bus to zero, regardless of the actual setting of the A20 address line on the processor. This component is in place to support legacy systems, but the QNX Neutrino OS doesn't require any such hardware. Note that some processors, such as the 386EX, have the A20 gate hardware built right into the processor itself — our IPL will disable the A20 gate as soon as possible after startup.

adaptive

Scheduling algorithm whereby a thread's priority is decayed by 1. See also **FIFO**, **round robin**, and **sporadic**.

adaptive partitioning

A method of dividing, in a flexible manner, CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

asymmetric multiprocessing (AMP)

A multiprocessing system where a separate OS, or a separate instantiation of the same OS, runs on each CPU.

atomic

Of or relating to atoms. :-)

In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered *indivisible*. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread's position.

attributes structure

Structure containing information used on a per-resource basis (as opposed to the **OCB**, which is used on a per-open basis).

This structure is also known as a **handle**. The structure definition is fixed (`iofunc_attr_t`), but may be extended. See also **mount structure**.

bank-switched

A term indicating that a certain memory component (usually the device holding an **image**) isn't entirely addressable by the processor. In this case, a hardware component manifests a small portion (or "window") of the device onto the processor's address bus. Special commands have to be issued to the hardware to move the window to different locations in the device. See also **linearly mapped**.

base layer calls

Convenient set of library calls for writing resource managers. These calls all start with *resmgr_*()*. Note that while some base layer calls are unavoidable (e.g. *resmgr_pathname_attach()*), we recommend that you use the **POSIX layer calls** where possible.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an "extension" to the BIOS or ROM Monitor — control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes 0x55 and 0xAA must be present (in that order) as the first two bytes in the device, with control being transferred to offset 0x0003.

block-integral

The requirement that data be transferred such that individual structure components are transferred in their entirety — no partial structure component transfers are allowed.

In a resource manager, directory data must be returned to a client as **block-integral** data. This means that only complete `struct dirent` structures can be returned — it's inappropriate to return partial structures, assuming that the next _IO_READ request will "pick up" where the previous one left off.

bootable

An image can be either bootable or **nonbootable**. A bootable image is one that contains the startup code that the IPL can transfer control to.

bootfile

The part of an OS image that runs the **startup code** and the Neutrino microkernel.

bound multiprocessing (BMP)

A multiprocessing system where a single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

budget

In **sporadic** scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for **`mkifs`** specifying the contents and other details of an **image**, or for **`mkefs`** specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or "cooked" mode. In this mode the character device library performs line-editing operations on each received character. Only when a line is "completely entered" — typically when a carriage return (CR) is received — will the line of data be made available to application processes. Contrast **raw mode**.

channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a **connection** (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using *ChannelCreate()*), and then receives messages from that channel (using *MsgReceive()*). Another thread that wishes to send a message to the first thread must make a connection to that channel by "attaching" to the channel (using *ConnectAttach()*) and then sending data (using *MsgSend()*).

chid

An abbreviation for **channel ID**.

CIFS

Common Internet File System (aka SMB) — a protocol that allows a client workstation to perform transparent file access over a network to a Windows 95/98/NT server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

CIS

Card Information Structure — a data block that maintains information about flash configuration. The CIS description includes the types of memory devices in the regions, the physical geometry of these devices, and the partitions located on the flash.

coid

An abbreviation for **connection ID**.

combine message

A resource manager message that consists of two or more messages. The messages are constructed as combine messages by the client's C library (e.g. *stat()*, *readblock()*), and then handled as individual messages by the resource manager.

The purpose of combine messages is to conserve network bandwidth and/or to provide support for atomic operations. See also **connect message** and **I/O message**.

connect message

In a resource manager, a message issued by the client to perform an operation based on a pathname (e.g. an `io_open` message). Depending on the type of connect message sent, a context block (see **OCB**) may be associated with the request and will be passed to subsequent I/O messages. See also **combine message** and **I/O message**.

connection

A kernel object used with message passing.

Connections are created by client threads to “connect” to the channels made available by servers. Once connections are established, clients can *MsgSendv()* messages over them. If a number of threads in a process all attach to the same channel, then the one connection is shared among all the threads. Channels and connections are identified within a process by a small integer.

The key thing to note is that connections and file descriptors (**FD**) are one and the same object. See also **channel** and **FD**.

context

Information retained between invocations of functionality.

When using a resource manager, the client sets up an association or **context** within the resource manager by issuing an *open()* call and getting back a file descriptor. The resource manager is responsible for storing the information required by the context (see **OCB**). When the client issues further file-descriptor based messages, the resource manager uses the OCB to determine the context for interpretation of the client’s messages.

cooked mode

See **canonical mode**.

core dump

A file describing the state of a process that terminated abnormally.

critical section

A code passage that *must* be executed “serially” (i.e. by only one thread at a time). The simplest from of critical section enforcement is via a **mutex**.

deadlock

A condition in which one or more threads are unable to continue due to resource contention. A common form of deadlock can occur when one thread sends a message to another, while the other thread sends a message to the first. Both threads are now waiting for each other to reply to the message. Deadlock can be avoided by good design practices or massive kludges — we recommend the good design approach.

device driver

A process that allows the OS and application programs to make use of the underlying hardware in a generic way (e.g. a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn't affect any other part of the OS — drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won't cause the entire OS to shut down.

discrete (or traditional) multiprocessor system

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

DNS

Domain Name Service — an Internet protocol used to convert ASCII domain names into IP addresses. In QNX native networking, `dns` is one of **Qnet**'s builtin resolvers.

dynamic bootfile

An OS image built on the fly. Contrast **static bootfile**.

dynamic linking

The process whereby you link your modules in such a way that the Process Manager will link them to the library modules before your program runs. The word "dynamic" here means that the association between your program and the library modules that it uses is done *at load time*, not at linktime. Contrast **static linking**. See also **runtime loading**.

edge-sensitive

One of two ways in which a **PIC** (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is "noticed" upon a transition to/from the rising/falling edge of a pulse. Contrast **level-sensitive**.

edited mode

See **canonical mode**.

EOI

End Of Interrupt — a command that the OS sends to the PIC after processing all Interrupt Service Routines (ISR) for that particular interrupt source so that the PIC can reset the processor's In Service Register. See also **PIC** and **ISR**.

EPROM

Erasable Programmable Read-Only Memory — a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g. 12V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. To change a bit from a 0 state into a 1 state can only be accomplished by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically 10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast **flash** and **RAM**.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requestor of the event.

FD

File Descriptor — a client must open a file descriptor to a resource manager via the *open()* function call. The file descriptor then serves as a handle for the client to use in subsequent messages. Note that a file descriptor is the exact same object as a connection ID (*coid*, returned by *ConnectAttach()*).

FIFO

First In First Out — a scheduling algorithm whereby a thread is able to consume CPU at its priority level without bounds. See also **adaptive**, **round robin**, and **sporadic**.

flash memory

A memory technology similar in characteristics to **EPROM** memory, with the exception that erasing is performed electrically instead of via ultraviolet light, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64k bytes at a time) instead of the entire device. Contrast **EPROM** and **RAM**.

FQNN

Fully Qualified NodeName — a unique name that identifies a QNX Neutrino node on a network. The FQNN consists of the nodename plus the node domain tacked together.

garbage collection

Aka space reclamation, the process whereby a filesystem manager recovers the space occupied by deleted files and directories.

HA

High Availability — in telecommunications and other industries, HA describes a system's ability to remain up and running without interruption for extended periods of time.

handle

A pointer that the resource manager base library binds to the pathname registered via *resmgr_attach()*. This handle is typically used to associate some kind of per-device information. Note that if you use the *iofunc_*()* **POSIX layer calls**, you must use a particular *type* of handle — in this case called an **attributes structure**.

hard thread affinity

A user-specified binding of a thread to a set of processors, done by means of a **runmask**. Contrast **soft thread affinity**.

image

In the context of embedded QNX Neutrino systems, an "image" can mean either a structure that contains files (i.e. an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim FFS-2-compatible filesystem (i.e. a flash filesystem image).

inherit mask

A bitmask that specifies which processors a thread's children can run on. Contrast **runmask**.

interrupt

An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.

interrupt handler

See **ISR**.

interrupt latency

The amount of elapsed time between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service routine. Also designated as "T_{il}". Contrast **scheduling latency**.

interrupt service routine

See **ISR**.

interrupt service thread

A thread that is responsible for performing thread-level servicing of an interrupt.

Since an **ISR** can call only a very limited number of functions, and since the amount of time spent in an ISR should be kept to a minimum, generally the bulk of the interrupt servicing work should be done by a thread. The thread attaches the interrupt (via *InterruptAttach()* or *InterruptAttachEvent()*) and then blocks (via *InterruptWait()*), waiting for the ISR to tell it to do something (by returning an event of type SIGEV_INTR). To aid in minimizing **scheduling latency**, the interrupt service thread should raise its priority appropriately.

I/O message

A message that relies on an existing binding between the client and the resource manager. For example, an _IO_READ message depends on the client's having previously established an association (or **context**) with the resource manager by issuing an *open()* and getting back a file descriptor. See also **connect message**, **context**, **combine message**, and **message**.

I/O privileges

A particular right, that, if enabled for a given thread, allows the thread to perform I/O instructions (such as the x86 assembler `in` and `out` instructions). By default, I/O privileges are disabled, because a program with it enabled can wreak havoc on a system. To enable I/O privileges, the thread must be running as `root`, and call *ThreadCtl()*.

IPC

Interprocess Communication — the ability for two processes (or threads) to communicate. QNX Neutrino offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), as well as signals.

IPL

Initial Program Loader — the software component that either takes control at the processor's reset vector (e.g. location 0xFFFFFFF0 on the x86), or is a BIOS extension. This component is responsible for setting up the machine into a usable state, such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also **BIOS extension signature** and **startup code**.

IRQ

Interrupt Request — a hardware request line asserted by a peripheral to indicate that it requires servicing by software. The IRQ is handled by the **PIC**, which then interrupts the processor, usually causing the processor to execute an **Interrupt Service Routine (ISR)**.

ISR

Interrupt Service Routine — a routine responsible for servicing hardware (e.g. reading and/or writing some device ports), for updating some data structures shared between the ISR and the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.

kernel

See **microkernel**.

level-sensitive

One of two ways in which a **PIC** (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast **edge-sensitive**.

linearly mapped

A term indicating that a certain memory component is entirely addressable by the processor. Contrast **bank-switched**.

message

A parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message — the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various "changes of state" that affect when, and for how long, they may run.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mount structure

An optional, well-defined data structure (of type `iofunc_mount_t`) within an *iofunc_*()* structure, which contains information used on a per-mountpoint basis (generally used only for filesystem resource managers). See also **attributes structure** and **OCB**.

mountpoint

The location in the pathname space where a resource manager has "registered" itself. For example, the serial port resource manager registers mountpoints for each serial device (**/dev/ser1**, **/dev/ser2**, etc.), and a CD-ROM filesystem may register a single mountpoint of **/cdrom**.

multicore system

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (*pthread_mutex_lock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a **critical section**). See also **critical section**.

name resolution

In a QNX Neutrino network, the process by which the **Qnet** network manager converts an **FQNN** to a list of destination addresses that the transport layer knows how to get to.

name resolver

Program code that attempts to convert an **FQNN** to a destination address.

nd

An abbreviation for **node descriptor**, a numerical identifier for a node *relative to the current node*. Each node's node descriptor for itself is 0 (ND_LOCAL_NODE).

NDP

Node Discovery Protocol — proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.

network directory

A directory in the pathname space that's implemented by the **Qnet** network manager.

Neutrino

Name of an OS developed by QNX Software Systems.

NFS

Network FileSystem — a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g. **ls**, **cp**, **mv**) operate on the remote files exactly as they do on your local files.

NMI

Nonmaskable Interrupt — an interrupt that can't be masked by the processor. We don't recommend using an NMI!

Node Discovery Protocol

See **NDP**.

node domain

A character string that the **Qnet** network manager tacks onto the nodename to form an **FQNN**.

nodename

A unique name consisting of a character string that identifies a node on a network.

nonbootable

A nonbootable OS image is usually provided for larger embedded systems or for small embedded systems where a separate, configuration-dependent setup may be required. Think of it as a second "filesystem" that has some additional files on it. Since it's nonbootable, it typically won't contain the OS, startup file, etc. Contrast **bootable**.

OCB

Open Control Block (or Open Context Block) — a block of data established by a resource manager during its handling of the client's *open()* function. This context block is bound by the resource manager to this particular request, and is then automatically passed to all subsequent I/O functions generated by the client on the file descriptor returned by the client's *open()*.

package filesystem

A virtual filesystem manager that presents a customized view of a set of files and directories to a client. The "real" files are present on some medium; the package filesystem presents a virtual view of selected files to the client.

partition

A division of CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

pathname prefix

See **mountpoint**.

pathname space mapping

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

persistent

When applied to storage media, the ability for the medium to retain information across a power-cycle. For example, a hard disk is a persistent storage medium, whereas a ramdisk is not, because the data is lost when power is lost.

Photon microGUI

The proprietary graphical user interface built by QNX Software Systems.

PIC

Programmable Interrupt Controller — hardware component that handles IRQs. See also **edge-sensitive**, **level-sensitive**, and **ISR**.

PID

Process ID. Also often *pid* (e.g. as an argument in a function call).

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface — the "X" alludes to "UNIX", on which the interface is based.

POSIX layer calls

Convenient set of library calls for writing resource managers. The POSIX layer calls can handle even more of the common-case messages and functions than the **base layer calls**. These calls are identified by the *iofunc_*()* prefix. In order to use these (and we strongly recommend that you do), you must also use the well-defined POSIX-layer **attributes** (`iofunc_attr_t`), **OCB** (`iofunc_ocb_t`), and (optionally) **mount** (`iofunc_mount_t`) structures.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been "preempted" by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY, the lower-priority thread is immediately preempted by the higher-priority thread.

prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent **priority inversion**.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU *on behalf of* the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it's working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one **thread** running in it — this thread is then called the first thread.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.

process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of 1.

pty

Pseudo-TTY — a character-based device that has two "ends": a master end and a slave end. Data written to the master end shows up on the slave end, and vice versa. These devices are typically used to interface between a program that expects a character device and another program that wishes to use that device (e.g. the shell and the `telnet` daemon process, used for logging in to a system over the Internet).

pulses

In addition to the synchronous Send/Receive/Reply services, QNX Neutrino also supports fixed-size, nonblocking messages known as pulses. These carry a small payload (four bytes of data plus a single byte code). A pulse is also one form of **event** that can be returned from an ISR or a timer. See *MsgDeliverEvent()* for more information.

Qnet

The native network manager in QNX Neutrino.

QoS

Quality of Service — a policy (e.g. `loadbalance`) used to connect nodes in a network in order to ensure highly dependable transmission. QoS is an issue that often arises in high-availability (**HA**) networks as well as realtime control systems.

RAM

Random Access Memory — a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast **flash** and **EPROM**.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data — you don't want any translations of the raw binary stream between the device and the application. Contrast **canonical mode**.

replenishment

In **sporadic** scheduling, the period of time during which a thread is allowed to consume its execution **budget**.

reset vector

The address at which the processor begins executing instructions after the processor's reset line has been activated. On the x86, for example, this is the address 0xFFFFFFF0.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX Neutrino resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g. serial ports, parallel ports, network cards, disk drives) or virtual (e.g. **/dev/null**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with **device drivers**. But unlike device drivers, QNX Neutrino resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also **device driver**.

RMA

Rate Monotonic Analysis — a set of methods used to specify, analyze, and predict the timing behavior of realtime systems.

round robin

Scheduling algorithm whereby a thread is given a certain period of time to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also **adaptive**, **FIFO**, and **sporadic**.

runmask

A bitmask that indicates which processors a thread can run on. Contrast **inherit mask**.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast **static linking**.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as "T_{sl}". Contrast **interrupt latency**.

scoid

An abbreviation for **server connection ID**.

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

soft thread affinity

The scheme whereby the microkernel tries to dispatch a thread to the processor where it last ran, in an attempt to reduce thread migration from one processor to another, which can affect cache performance. Contrast **hard thread affinity**.

software interrupts

Similar to a hardware interrupt (see **interrupt**), except that the source of the interrupt is software.

sporadic

Scheduling algorithm whereby a thread's priority can oscillate dynamically between a "foreground" or normal priority and a "background" or low priority. A thread is given an execution **budget** of time to be consumed within a certain **replenishment** period. See also **adaptive**, **FIFO**, and **round robin**.

startup code

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

static bootfile

An image created at one time and then transmitted whenever a node boots. Contrast **dynamic bootfile**.

static linking

The process whereby you combine your modules with the modules from the library to form a single executable that's entirely self-contained. The word "static" implies that it's not going to change — *all* the required modules are already combined into one.

symmetric multiprocessing (SMP)

A multiprocessor system where a single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under QNX Neutrino. A thread is a flow of execution; it exists within the context of a **process**.

tid

An abbreviation for **thread ID**.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an **event**, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a **round-robin** or **adaptive** scheduled thread. This period of time is small (on the order of tens of milliseconds); the actual value shouldn't be relied upon by any program (it's considered bad design).

Index

!

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

T

U

V

W

X

Z