QNX® Neutrino® RTOS

User's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 06, 2020

Contents

About the QNX Neutrino User's Guide

The QNX Neutrino *User's Guide* is intended for *all* users of a QNX Neutrino RTOS system, from system administrators to end users.

This guide tells you how to:

- Use the QNX Neutrino *runtime* environment, regardless of the kind of computer it's running on (embedded system or desktop). Think of this guide as the companion how-to doc for the *Utilities Reference*. Assuming there's a system prompt waiting for input, this guide is intended to help you learn how to interact with that prompt.
- Perform such traditional system administration topics as setting up user accounts, security, starting up a QNX Neutrino machine, etc.

This *User's Guide* is intended for programmers who develop QNX Neutrino-based applications, as well as OEMs and other "resellers" of the OS, who may want to pass this guide on to their end users as a way to provide documentation for the OS component of their product.

- Your system might not include all of the things that this guide describes, depending on what software you've installed. For example, some utilities are included with the OS, and others are included in a specific Board Support Package (BSP).

  The online version of this guide contains links to various books throughout our entire documentation set; if you don't have the entire set installed on your system, you'll naturally get some bad-link errors (e.g., "File not found").
- Disable **PnP-aware OS** in the BIOS.

The following table may help you find information quickly:

To find out about:	Go to:
How QNX Neutrino compares to other operating systems	*Getting to Know the OS*
Starting and ending a session, and turning off a QNX Neutrino system	*Logging In, Logging Out, and Shutting Down*
Adding users to the system, managing passwords, etc.	*Managing User Accounts*
The basics of using the keyboard, command line, and shell (command interpreter)	*Using the Command Line*
Files, directories, and permissions	*Working with Files*
How to edit files	*Using Editors*
Customizing your shell, setting the time, etc.	*Configuring Your Environment*
Creating your own commands	*Writing Shell Scripts*
The filesystems that QNX Neutrino supports	*Working with Filesystems*

To find out about:	Go to:
Accessing other machines with QNX Neutrino's native networking	*Using Qnet for Transparent Distributed Processing*
Setting up TCP/IP	*TCP/IP Networking*
Backing up and restoring your files	*Backing Up and Recovering Data*
Making your QNX Neutrino system more secure	*Securing Your System*
Analyzing and improving your machine's performance	*Fine-Tuning Your System*
How many processes, files, etc. your system can support	*Understanding System Limits*
How to get help	*Technical Support*
Samples of buildfiles, profiles, etc.	*Examples*
Terms used in this document	*Glossary*

For information about programming in QNX Neutrino, see *Getting Started with QNX Neutrino* and the QNX Neutrino *Programmer's Guide*.

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`
Constants	NULL
Data types	`unsigned short`
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

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.

DANGER: 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 typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

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

Chapter 1
Getting to Know the OS

Welcome to the QNX Neutrino RTOS!

This section describes how the QNX Neutrino RTOS compares to Unix and Microsoft Windows, from a user's (not a developer's) perspective. For more details about QNX Neutrino's design and the philosophy behind it, see the *System Architecture* guide.

QNX Neutrino compared with Unix

If you're familiar with Unix-style operating systems, you'll feel right at home with QNX Neutrino—many people even pronounce “QNX” to rhyme with “Unix” (some spell it out: Q-N-X).

At the heart of the system is the microkernel, `procnto`, surrounded by other processes and the familiar Korn shell, `ksh` (see the *Using the Command Line* chapter). Each process has its own process ID, or *pid*, and contains one or more threads.

To determine the release version of the kernel on your system, use the `uname -a` command. For more information, see its entry in the *Utilities Reference*.

QNX Neutrino is a multiuser OS; it supports any number of users at a time. The users are organized into groups that share similar permissions on files and directories. For more information, see *Managing User Accounts*.

QNX Neutrino follows various industry standards, including POSIX (shell and utilities) and TCP/IP. This can make porting existing code and scripts to QNX Neutrino easier.

QNX Neutrino's command line looks just like the Unix one; QNX Neutrino supports many familiar utilities (`grep`, `find`, `ls`, `gawk`) and you can connect them with pipes, redirect the input and output, examine return codes, and so on. Many utilities are the same in Unix and QNX Neutrino, but some have a different name or syntax in QNX Neutrino:

Unix	QNX Neutrino	See also:
`adduser`	`passwd`	*Managing User Accounts*
`at`	`cron`	
`dmesg`	`slogger2`, `slog2info`	
`fsck`	`chkqnx6fs`, `chkdosfs`	
`ifconfig eth0`	`ifconfig en0`	*TCP/IP Networking*
`man`	`use`	*Using the Command Line*
`pg`	`less`, `more`	*Using the Command Line*

For details on each command, see the QNX Neutrino *Utilities Reference*.

QNX Neutrino compared with Microsoft Windows

QNX Neutrino and Windows have different architectures, but the main difference between them from a user's perspective is how you invoke programs.

Much of what you do via a GUI in Windows you do in QNX Neutrino through command-line utilities, configuration files, and scripts, although QNX Neutrino does support a powerful Integrated Development Environment (IDE) to help you create, test, and debug software and embedded systems.

Here are some other differences:

- QNX Neutrino and DOS use different end-of-line characters; QNX Neutrino uses a linefeed, while DOS uses a carriage return and a linefeed. If you need to transfer text files from one OS to the other, you can use QNX Neutrino's `textto` utility to convert the files. For example, to convert the end-of-line characters to QNX Neutrino-style:

  ```
  textto -l my_file
  ```

  To convert the end-of-line characters to DOS-style:

  ```
  textto -c my_file
  ```

- QNX Neutrino uses a slash (/) instead of a backslash (\) to separate components of a pathname.

How QNX Neutrino is unique

The QNX Neutrino RTOS consists of a microkernel (`procnto`) and various processes. Each process—even a device driver—runs in its own virtual memory space.

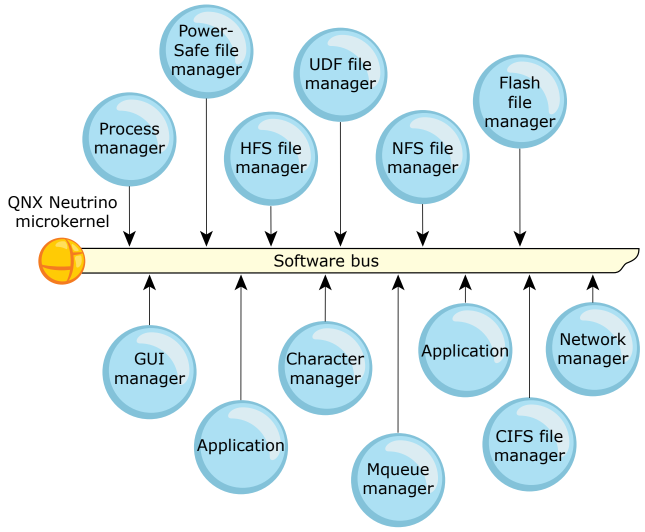


Figure 1: The QNX Neutrino architecture.

The advantage of using virtual memory is that one process can't corrupt another process's memory space. For more information, see The Philosophy of QNX Neutrino in the *System Architecture* guide.

QNX Neutrino's most important features are its microkernel architecture and the resource manager framework that takes advantage of it (for a brief introduction, see "*Resource managers*"). Drivers have exactly the same status as other user applications, so you debug them using the same high-level, source-aware, breakpointing IDE that you'd use for user applications. This also means that:

- You aren't also debugging the kernel when you're debugging a driver.
- A faulty driver isn't likely to crash the OS.
- You can usually stop and restart a driver without rebooting the system.

Developers can usually eliminate interrupt handlers (typically the most tricky code of all) by moving the hardware manipulation code up to the application thread level—with all the debugging advantages and freedom from restrictions that that implies. This gives QNX Neutrino an enormous advantage over monolithic systems.

Likewise, in installations in the field, the modularity of QNX Neutrino's components allows for the kind of redundant coverage expressed in our simple, yet very effective, High Availability (HA) manager, making it much easier to construct extremely robust designs than is possible with a more fused approach. People seem naturally attracted to the ease with which functioning devices can be planted in the POSIX pathname space as well.

Developers, system administrators, and users also appreciate QNX Neutrino's adherence to POSIX, the realtime responsiveness that comes from our devotion to short nonpreemptible code paths, and the general robustness of the microkernel.

CAUTION: Some x86 systems can run in System Management Mode (SMM), where the BIOS installs special code that runs when a System Management Interrupt (SMI) occurs. SMI interrupts may be generated by the motherboard or peripheral hardware, and can't be masked by the operating system. When SMM is entered, normal operations—including the OS—are suspended, and the SMI handler runs at a high priority. Avoid using systems where SMM can't be disabled, because it can destroy QNX Neutrino's realtime performance. The OS can't do anything about the delays that SMM introduces, nor can the OS even detect that the system has entered SMM.

QNX Neutrino's microkernel architecture lets developers scale the code down to fit in a very constrained embedded system, but QNX Neutrino is powerful enough to use as a desktop OS. QNX Neutrino runs on multiple platforms, including x86, and ARM. It supports symmetric multiprocessing (SMP) and bound multiprocessing (BMP) on multicore systems with up to 32 processors; for more information, see the Multicore Processing chapter of the QNX Neutrino *Programmer's Guide*.

QNX Neutrino also features the Qnet protocol, which provides transparent distributed processing; you can access the files or processes on any machine on your network as if they were on your own machine.

Resource managers

A *resource manager* is a server program that accepts messages from other programs and, optionally, communicates with hardware. All of the QNX Neutrino device drivers and filesystems are implemented as resource managers.

QNX Neutrino resource managers are responsible for presenting an interface to various types of devices. This may involve managing actual hardware devices (such as serial ports, parallel ports, network cards, and disk drives) or virtual devices (such as **/dev/null**, the network filesystem, and pseudo-ttys).

The binding between the resource manager and the client programs that use the associated resource is done through a flexible mechanism called *pathname-space mapping*. In pathname-space mapping, an association is made between a pathname and a resource manager. The resource manager sets up this mapping by informing the QNX Neutrino process manager that it's responsible for handling requests at (or below, in the case of filesystems), a certain mountpoint. This allows the process manager to associate services (i.e., functions provided by resource managers) with pathnames.

Once the resource manager has established its pathname prefix, it receives messages whenever any client program tries to do an *open()*, *read()*, *write()* , etc. on that pathname.

For more detailed information on the resource manager concept, see Resource Managers in *System Architecture*.

Chapter 2
Logging In, Logging Out, and Shutting Down

QNX Neutrino is a *multiuser* operating system; it lets multiple users log in and use the system simultaneously, and it protects them from each other through a system of resource ownership and permissions.

Depending on the configuration, your system boots into text mode and prompts you for your user ID and password.

Your system might have been configured so that you don't have to log in at all.

`root` or non-`root`?

The QNX Neutrino RTOS recognizes user ID 0 as being privileged, and traditionally an account with uid 0 is called **root**.

This user can do anything on the system; it has what Windows calls “administrator's privileges”. Unix-style operating systems often call **root** the “superuser”. For information about creating a non-**root** account, see *Managing User Accounts*.

Logging in

If your system is configured to do so, it prompts you for your user name and then your password.

The system does this by automatically starting the `login` utility.

If you type an invalid user name, the system prompts you for the password anyway. This avoids giving clues to anyone who's trying to break into the system.

Text mode on an x86 machine could be on a physical console supplied by `devc-con` or `devc-con-hid`. On any other type of machine, you could be connecting to the target via a serial port or TCP/IP connection.

Once you've logged in

After you've logged in, the system automatically runs the **/home/***username***/.profile** script.

This script lets you customize your working environment without affecting other users. For more information, see *Configuring Your Environment*.

To change your password:

Use the `passwd` command. This utility prompts you for your current and new passwords; see “*Managing your own account*” in Managing User Accounts.

To log in as a different user:

Enter `login` at the command prompt, and then enter the user's name and password.

The `su` (switch user ID) utility also lets you run as another user, but temporarily. It doesn't run the user's profiles or significantly modify the environment. For more information, see the *Utilities Reference*.

To determine your current user name:

Use the `id` command.

Logging out

To log out, type `logout` at the command prompt. You can also log out by terminating your login shell; just enter the `exit` shell command or press **Ctrl–D**.

Shutting down and rebooting

You rarely need to reboot a QNX Neutrino system. If a driver or other system process crashes, you can usually restart that one process.

Don't simply turn off a running QNX Neutrino system, because processes might not shut down properly, and any data that's in a filesystem's cache might not get written to the disk.

To shut down or reboot the system, use the `shutdown` command. You can do this only if you're logged in as **root**. This utility has several options that let you:

- name the node to shut down (default is the current node)
- specify the type of shutdown (default is to reboot)
- shut down quickly
- list the actions taken while shutting down (i.e., be verbose)

Before the shutdown program shuts down the system, it sends a SIGTERM signal to any running processes, to give them the opportunity to terminate cleanly. For more information, see the *Utilities Reference*.

Chapter 3
Managing User Accounts

This chapter explains how user accounts work, how users can change their password by using the `passwd` utility, and how system administrators can use the `passwd` utility and edit account database files to create and maintain users' accounts.

In embedded systems, the designer may choose to eliminate the account-related files from the system, disabling logins and references to users and groups by name, even though the system remains fully multiuser and may have multiple numeric user IDs running programs and owning system resources. If your system is configured this way, most of this chapter won't be relevant to you.

What does a user account do?

A user account associates a textual user name with a numeric user ID and group ID, a login password, a user's full name, a home directory, and a login shell. This data is stored in the **/etc/passwd** and **/etc/shadow** files, where it's accessed by login utilities as well as by other applications that need user-account information.

User names and passwords are case-sensitive.

User accounts let:

- users log in with a user name and password, starting a session under their user ID and group ID
- users create their own login environments
- applications determine the user name and account information relating to a user ID and group ID if they're defined in **/etc/passwd** and **/etc/group** (e.g., `ls -l` displays the names—not the IDs—of the user and group who own each file)
- utilities and applications accept user names as input as an alternative to numeric user IDs
- shells expand ~*username* paths into actual pathnames, based on users' home directory information stored in their accounts

Groups are used to convey similar permissions to groups of users on the system. Entries in **/etc/passwd** and **/etc/group** define group membership, while the group ID of a running program and the group ownership and permission settings of individual files and directories determine the file permission granted to a group member.

When you log in, you're in the group specified in **/etc/passwd**. You can switch to another of your groups by using the `newgrp` utility.

User accounts vs user IDs: login, lookup, and permissions

Once you've logged in, the *numeric* user ID of your running programs and system resources determines your programs' ability to access resources and perform operations, such as sending signals to other processes. Textual names are used only by utilities and applications that need to convert between names and numeric IDs.

Changing user names, groups, user IDs, and so on in the account database has no effect on your permission to access files, etc. until you next log in.

The **root** user (user ID 0) has permission to do nearly anything to files, regardless of their ownership and permission settings. For more information, see "*File ownership and permissions*" in Working with Files.

When the shell interprets a ~*username* pathname, it gets the user's home directory from **/etc/passwd**. If you remove or change a user's account, any shell running in the system that had previously accessed that user's home directory via ~*username* may be using the old home directory information to determine the actual path, because the shell caches the data.

New shells read the data afresh from **/etc/passwd**. This may be a problem if a shell script that uses ~*username* invokes another shell script that also uses this feature: the two scripts would operate on different paths if the home directory information associated with the user name has changed since the first shell looked the information up.

What happens when you log in?

You typically start a session on the computer by logging in; the configuration of your account determines what happens then.

When you log in, the system creates a user session led by a process that runs under your user ID and default group ID, as determined from your account entry in **/etc/passwd**.

The user ID and group ID determine the permission the process has to access files and system resources. In addition, if the process creates any files and directories, they belong to that user and group. Each new process that you start inherits your user ID and group ID from its parent process. For more information about file permissions, see "*File ownership and permissions*" in Working with Files.

For more information on characteristics that programs inherit from their parent programs, see *spawn()* in the QNX Neutrino *C Library Reference*. For more information on sessions and process groups, see IEEE Std 1003.1-2001 *Standard for Information Technology Portable Operating System Interface*.

When you log in via the `login` utility, `login` changes directory to your *HOME* directory; it also sets *LOGNAME* to your user name and *SHELL* to the login shell named in your account. It then starts the login shell, which is typically a command interpreter (**/bin/sh**), but could also be an application that gets launched as soon as you log in.

Account database

The account database consists of the files (listed with the appropriate access permissions) described in this section.

File:	Owner:	Group:	Permissions:
/etc/passwd	**root**	**root**	`rw- r-- r--`
/etc/group	**root**	**root**	`rw- r-- r--`
/etc/shadow	**root**	**root**	`rw- --- ---`
/etc/.pwlock	**root**	**root**	`rw- r-- r--`

Note that anyone can read **/etc/passwd**. This lets standard utilities find information about users. The hashed password isn't stored in this file; it's stored in **/etc/shadow**, which only **root** has permission to read. This helps prevent attempts to decrypt the passwords.

To protect the security of your user community, make sure you don't change these permissions.

`/etc/passwd`

This file stores information about users.

Each line in **/etc/passwd** is in this format:

username **:** *has_pw* **:** *userid* **:** *group* **:** *comment* **:** *homedir* **:** *shell*

The fields are separated by colons and include:

username

The user's login name. This can contain any characters except a colon (`:`), but you should probably avoid any of the shell's special characters. For more information, see "*Quoting special characters*" in Using the Command Line.

has_pw

This field must be empty or `x`. If empty, the user has no password; if `x`, the user's hashed password is in **/etc/shadow**.

userid

The numeric user ID.

group

The numeric group ID.

comment

A free-form comment field that usually contains at least the user's real name; this field must not contain a colon.

homedir

The user's home directory.

shell

The initial command to start after `login`. The default is **/bin/sh**.

You can't specify any arguments to the login program.

Here's a sample entry from **/etc/passwd**:

```
fred:x:290:120:Fred L. Jones:/home/fred:/bin/sh
```

/etc/group

This file stores information about the groups on your system.

Each line in **/etc/group** is in this format:

groupname`:x:`*group_ID*`:[`*username*`[,`*username*`]...]`

The fields are separated by colons and include:

groupname

The name of the group. Like a user's name, this can contain any characters except a colon (`:`), but you should probably avoid any of the shell's special characters. For more information, see "*Quoting special characters*" in Using the Command Line.

`x`

The password for the group. QNX Neutrino doesn't support group passwords.

group_ID

The numeric group ID.

***username*[,*username*]...**

The user names of the accounts that belong to this group, separated by commas (`,`).

Here's a sample entry:

```
techies:x:123:michel,ali,sue,jake
```

/etc/shadow

This file stores hashed passwords and other account and password information.

Each line in **/etc/shadow** is in the following format:

username **:** *password* **:** *last_change* **:** *min_age* **:** *max_age* **:** *warning* **:** *inactivity* **:** *expiry* **:** *reserved*

username

The user's login name. It must be a valid username that exists on the system.

password

The user's hashed password.

If the password field contains a string that is not a valid hash of the password, the user cannot log in.

This field can be empty. When it is empty, it may be possible to log in as the corresponding user without specifying a password, depending on the application that performs the authentication or the configuration of the PAM module that performs the authentication.

If the password field starts with an exclamation mark, the password is locked. The characters that follow the exclamation mark are the password hash before the password was locked.

The password is captured in one of the following formats:

- *@digest@hash@salt*
- *@digest,iterations@hash@salt*

where:

digest is a single character that indicates which digest function was used to hash the password. The following values are currently allowed: `s` (sha256), `S` (sha512).

interations is the number of iterations to perform during key derivation. If not specified, the default value of 4096 is used.

hash is the Base64-encoded hashed password.

salt is the Base64-encoded salt value.

last_change

The date of the last password change, expressed as the number of days since the start of the Unix Epoch.

min_age

The minimum number of days the user must wait after changing the password before he or she can change it again.

An empty field or `0` specifies that no wait time is required.

max_age

When this number of days has passed after a password change, the user must change his or her password again. After this number of days has passed, the user is prompted to change the password the next time he or she logs in.

An empty field or a value of `0` specifies that there is no maximum number of days after which the user must change his or her password.

warning

The number of days before the password is set to expire that users are warned that they must change their password.

An empty field or a value of `0` specifies that no warning is given.

inactivity

The password expiration date. Not currently implemented.

expiry

The account expiration date, expressed as the number of days since the Unix Epoch.

An empty field or a value of `0` specifies that the account never expires.

reserved

Reserved for future use.

/etc/.pwlock

The `passwd` utility creates **/etc/.pwlock** to indicate to other instances of `passwd` that the password file is currently being modified. When `passwd` finishes, it removes the lock file.

If you're the system administrator, and you need to edit the account files, you should:

1. Lock the password database: if the **/etc/.pwlock** file doesn't exist, lock the account files by creating it; if it does exist, wait until it's gone.
2. Open the appropriate file or files, using the text editor of your choice, and make the necessary changes.
3. Unlock the password database by removing **/etc/.pwlock**.

Managing your own account

As a regular (non-**root**) user, you can change your own password. You can also customize your environment by modifying the configuration files in your home directory; see the Configuring Your Environment chapter.

Changing your password

To change your password, use the `passwd` utility. You're prompted for your current password and then for a new one. You have to repeat the new password to guard against typographical errors.

Depending on the password rules that the system administrator has set, `passwd` may require that you enter a password of a certain length or one that contains certain elements (such as a combination of letters, numbers, and punctuation). If the password you select doesn't meet the criteria, `passwd` asks you to choose another.

If other users can access your system (e.g., it's connected to the Internet, has a dial-in modem, or is physically accessible by others), be sure to choose a password that will secure your account from unauthorized use. You should choose passwords that:

- are more than 5 characters long
- consist of multiple words or numbers and include punctuation or white space
- you haven't used on other systems (many systems, and websites in particular, don't store and communicate passwords in a hashed form; this lets people who gain access to those systems see your password in plain text)
- incorporate both uppercase and lowercase letters
- don't contain words, phrases, or numbers that other people can guess (e.g., avoid the names of family members and pets, license plate numbers, and birthdays)

For more information on system security, see *Securing Your System*.

Forgot your password?

If you forget your password, ask the system administrator (**root** user) to assign a new password to your account. Only **root** can do this.

In general, no one can retrieve your old password from the **/etc/shadow** file. If your password is short or a single word, your system administrator—or a hacker—can easily figure it out, but you're better off with a new password.

If you're the system administrator, and you've forgotten the password for **root**, you need to find an alternate way to access the **/etc/passwd** and **/etc/shadow** files in order to reset the **root** password. Some possible ways to do this are:

- Boot the system from another disk or device where you can log in as **root**, and from there manually reset the password.
- Access the necessary files from the **root** account of another QNX Neutrino machine, using Qnet. For more information, see *Using Qnet for Transparent Distributed Processing*.

- Remove the media on which the **/etc/passwd** and **/etc/shadow** are stored and install it on another QNX Neutrino machine from which you can modify the files.
- In the case of an embedded system, build a new image that contains new **passwd** and **shadow** files, and then transfer it to your target system.

Managing other accounts

As a system administrator, you need to add and remove user accounts and groups, manage passwords, and troubleshoot users' problems. You must be logged in as **root** to do this, because other users don't have permission to modify **/etc/passwd**, **/etc/shadow**, and **/etc/group**.

CAUTION: While it's safe at any time to use the `passwd` utility to change the password of an existing user who already has a password, it isn't necessarily safe to make any other change to the account database while your system is in active use. Specifically, the following operations may cause applications and utilities to operate incorrectly when handling user-account information:

- adding a user, either by using the `passwd` utility or by manually editing **/etc/passwd**
- putting a password on an account that previously didn't have a password
- editing the **/etc/passwd** or **/etc/group** files

If it's likely that someone might try to use the `passwd` utility or update the account database files while you're editing them, lock the password database by creating the **/etc/.pwlock** file before making your changes.

As described below, you should use the `passwd` utility to change an account's password. However, you need to use a text editor to:

- change an existing user's user name, full name, user ID, group ID, home directory, or login shell
- create a new account that doesn't conform to the `passwd` utility's allowed configuration
- remove a user account
- add or remove a group
- change the list of members of a group

The changes you make manually to the account files aren't checked for conformance to the rules set in the `passwd` configuration file. For more information, see the description of **/etc/default/passwd** in the documentation for `passwd` in the *Utilities Reference*.

Adding users

To add a user:

1. Log in as **root**.
2. Use `passwd`:

   `passwd` *new_username*

   Make sure that the user name is no longer than 14 characters; otherwise, that user won't be able to log in.

If you specify a user name that's already registered, `passwd` assumes you want to change their password. If that's what you want, just type in the new password and then confirm it. If you don't wish to change the user's password, press **Ctrl–C** to terminate the `passwd` utility without changing anything.

If the user name isn't already registered, `passwd` prompts you for account information, such as the user's group list, home directory, and login shell. The **/etc/default/passwd** configuration file specifies the rules that determine the defaults for new accounts. For more information, see the description of this file in the documentation for `passwd`.

The prompts include:

`User id # (`*default*`)`

Specify the numeric user ID for the new user. By default, no two users may share a common user ID, because applications won't be able to determine the user name that corresponds to that user ID.

`Group id # (`*default*`)`

Choose a numeric group ID that the user will belong to after initially logging in.

The `passwd` utility doesn't add the new user to the group's entry in the **/etc/group** file; you need to do that manually using a text editor. See “*Defining Groups*” for more details.

`Real name ()`

Enter the user's real name. The real name isn't widely used by system utilities, but may be used by applications such as email.

`Home directory (/home/`*username*`)`

Enter the pathname of the user's home directory, usually **/home/***username*. The `passwd` utility automatically creates the directory you specify. If the directory already exists, `passwd` by default prompts you to select a different pathname. For information on disabling this feature, see the description of **/etc/default/passwd** in the documentation for `passwd`.

`Login shell (/bin/sh)`

This is the program that's run once the user logs in. Traditionally, this is the shell (`/bin/sh`), giving the user an interactive command line upon logging in.

You can specify any program as the login shell, but you can't pass command-line arguments to it.

Instead of specifying a custom program within the account entry, you should customize the user's **.profile** file in their home directory; `/bin/sh` runs this profile automatically when it starts up. For more information, see *Configuring Your Environment*.

`New password:`

Specify the initial password for the account. You're asked to confirm it by typing it again.

Removing accounts

To remove a user account:

1. Lock the user account database: if the **/etc/.pwlock** file doesn't exist, lock the account files by creating it; if it does exist, wait until it's gone.
2. Remove the account entry in **/etc/passwd** and **/etc/shadow** to disable future logins, or change the login shell to a program that simply terminates, or that displays a message and then terminates.
3. Remove references to the user from the **/etc/group** file.
4. Unlock the account database by removing **/etc/.pwlock**.
5. If necessary, remove or change ownership of system resources that the user owned.
6. If necessary, remove or alter references to the user in email systems, TCP/IP access control files, applications, and so on.

Instead of removing a user, you can disable the account by using the `passwd` utility to change the account's password. In this way, you can tell which system resources the former user owned, since the user ID-to-name translation still works. When you do this, the `passwd` utility automatically handles the necessary locking and unlocking of the account database.

If you ever need to log in to that account, you can either use the `su` ("switch user") utility to switch to that account (from **root**), or log in to the account. If you forget the password for the account, remember that the **root** user can always change it.

What should you do with any resources that a former user owned? Here are some of your options:

- If you've retained the user account in the account database but disabled it by changing the password or the login shell, you can leave the files as they are.
- You can assign the files to another user:

  ```
  find / -user user_name_or_ID -chown new_username
  ```

- You can archive the files, and optionally move them to other media:

  ```
  find / -user user_name_or_ID |  pax -wf archivefile
  ```

- You can remove them:

  ```
  find / -user user_name_or_ID -remove!
  ```

CAUTION: If you remove a user's account in the account database but don't remove or change the ownership of their files, it's possible that a future account may end up with the same numeric user ID, which would make the new user the owner of any files left behind by the old one.

Defining groups

A user's account entry in **/etc/passwd** solely determines which group the user is part of on logging in, while the groups a user is named in within the **/etc/group** file solely determine the groups the user may switch to after logging in (see the `newgrp` utility). As with user names and IDs, the *numeric* effective group ID of a running program determines its access to resources.

For example, if you have a team of people that require access to **/home/projects** on the system, but you don't want the other users to have access to it, do the following:

1. Add a group called **projects** to the **/etc/group** file, adding all necessary users to that group (for details, see "*Creating a new group*," below).
2. If you want this group to be the default for these users, change their account entries in **/etc/passwd** to reflect their new default group ID.
3. Recursively change the group ownership and permissions on **/home/projects**:

   ```
   chgrp -R projects /home/projects
   chmod -R g+rw /home/projects
   ```

4. Remove access for all other users:

   ```
   chmod -R o-rwx /home/projects
   ```

For more details on permissions, see "*File ownership and permissions*" in Working with Files.

Creating a new group

To create a new group, open **/etc/group** in a text editor, then add a line that specifies the new group's name, ID, and members.

For example:

```
techies:x:101:michel,jim,sue
```

For more information about the fields, see "*/etc/group*," earlier in this chapter.

CAUTION: Do this work at a time when the system is idle. As your text editor writes the **/etc/group** file back, any application or utility that's trying to simultaneously read the **/etc/group** file (e.g `ls -l`, `newgrp`) might not function correctly.

Modifying an existing group

Each time you add a new user to a group (e.g., when you use `passwd` to create a new user account), you need to edit the **/etc/group** file and add the user to the appropriate group entry.

For instance, if you have an existing group `techies` and want to add zeke to the group, change:

```
techies:x:101:michel,jim,sue
```

to:

```
techies:x:101:michel,jim,sue,zeke
```

You should do this at a time when you're certain no users or programs are trying to use the **/etc/group** file.

Troubleshooting

Here are some problems you might encounter while working with passwords and user accounts.

The* `passwd` *utility seems to hang after I change my password.

The `passwd` utility uses the **/etc/.pwlock** file as a lock while updating the password database. If the file already exists, `passwd` won't run.

If the system crashes during the update, and **/etc/.pwlock** still exists, `passwd` refuses to work until the system administrator removes the file.

If the password files are left in an inconsistent state as a result of the crash, the system administrator should also copy the backup files, **/etc/oshadow** and **/etc/opasswd**, to **/etc/shadow** and **/etc/passwd** to prevent additional problems.

Why can't I log in?

If you enter your user name and password to the login prompt, `login`, and it responds `Login incorrect`, it's likely because your user name doesn't exist, or you've typed the wrong password. Both user names and passwords are case-sensitive; make sure you don't have **Caps Lock** on.

To avoid giving clues to unauthorized users, `login` doesn't tell you whether it's the user name or the password that's wrong. If you can't resolve the problem yourself, your system administrator (**root** user) can set a new password on your account.

This symptom can also occur if one or more password-related files are missing. If the system administrator is in the middle of updating the files, it's possible that its absence will be temporary. Try again in a minute or two if this might be the case. Otherwise, see your system administrator for help.

If you *are* the system administrator and can't access the system, try accessing it from another QNX Neutrino machine using Qnet, or from a development host using the `qconn` interface.

***My login fails with a message: command* `: No such file or directory`.**

The system couldn't find the command specified as your login shell. This might happen because:

- The command wasn't found in `login`'s *PATH* (usually `/bin:/usr/bin`). Specify the full pathname to the program (e.g., **/usr/local/bin/myprogram**) in the user's **/etc/passwd** account entry.
- The account entry specifies options or arguments for your login shell. You can't pass arguments to the initial command, because the entire string is interpreted as the filename to be executed.

Chapter 4
Using the Command Line

Like QNX 4, Unix, and DOS, the QNX Neutrino RTOS is based on a command-line interface that you might want or need to use instead of the GUI.

For developing software, you don't always have to use the command line; on Linux and Windows, you can use our Integrated Development Environment (IDE) that provides a graphical way to write, build, and test code. The IDE frequently uses QNX Neutrino utilities, but “hides” the command line from you. For more information, see the IDE *User's Guide*.

Processing a command

When you type a command, several different processes interpret it in turn.

1. The driver for your character device interprets such keys as **Backspace** and **Ctrl–C**.
2. The *command interpreter* or *shell* breaks the command line into tokens, interprets them, and then invokes any utilities.
3. The utilities parse the command line that the shell passes to them, and then they perform the appropriate actions.

Character-device drivers

When you type a command, the first process that interprets it is the character-device driver.

The driver that you use depends on your hardware; for more information, see the entries for the `devc-*` character I/O drivers in the *Utilities Reference*.

Some keys may behave differently from how they're described here, depending on how you configure your system.

For more information, see Character I/O in the *System Architecture* guide.

Input modes

Character-device drivers run in either *raw input mode*, or *canonical* (or *edited input*) mode.

In raw input mode, each character is submitted to an application process as it's received; in edited input mode, the application process receives characters only after a whole line has been entered (usually signalled by a carriage return).

Terminal support

Some programs, such as `vi`, need to know just what your terminal can do, so that they can move the cursor, clear the screen, and so on. The *TERM* environment variable indicates the type of terminal that you're using, and the **/usr/lib/terminfo** directory is the terminal database.

In this directory, you can find subdirectories (**a** through **z**) that contain the information for specific terminals. Some applications use **/etc/termcap**, the older single-file database model, instead of **/usr/lib/terminfo**.

The default terminal is **qansi-m**, the QNX version of an ANSI terminal. For more information about setting the terminal type, see "*Terminal types*" in Configuring Your Environment.

Telnet

If you're using `telnet` to communicate between two QNX machines (QNX 4, QNX Neutrino), use the `-8` option to enable an eight-bit data path. If you're connecting to a QNX Neutrino box from some other operating system, and the terminal isn't behaving properly, quit from `telnet` and start it again with the `-8` option.

To `telnet` from Windows to a QNX Neutrino machine, use `ansi` or `vt100` for your terminal type.

The keyboard at a glance

The table below describes how the character-device drivers interpret various keys and keychords (groups of keys that you press simultaneously). The drivers handle these keys as soon as you type them.

Your keyboard might not behave as indicated if:

- The driver is in raw input mode instead of edited input mode.
- You're working with an application that has complex requirements for user interaction (e.g., the application might take control over how the keyboard works).

  or:

- You're working at a terminal that has keyboard limitations.

If you want to:	Press:
Move the cursor to the left	(left arrow)
Move the cursor to the right	(right arrow)
Move the cursor to the start of a line	**Home**
Move the cursor to the end of a line	**End**
Delete the character left of the cursor	**Backspace**
Delete the character at the cursor	**Del**
Delete all characters on a line	**Ctrl–U**
Toggle between insert and typeover modes (if an application supports them)	**Ins**
Submit a line of input or start a new line	**Enter**
Recall a command (see below)	or (up or down arrow)
Suspend the displaying of output	**Ctrl–S**
Resume the displaying of output	**Ctrl–Q**
Attempt to kill a process	**Ctrl–C** or **Ctrl–Break**
Indicate end of input (EOF)	**Ctrl–D**
Clear the terminal	**Ctrl–L**

When you use the up or down arrow, the character-device driver passes a "back" or "forward" command to the *shell*, which recalls the actual command.

Physical and virtual consoles

The display adapter, the screen, and the system keyboard are collectively referred to as the *physical console*, which is controlled by a console driver.

Some systems don't include a console driver. For example, embedded systems might include only a serial driver (`devc-ser*`). The `devc-con` and `devc-con-hid` drivers are currently supported only on x86 platforms.

To let you interact with several applications at once, QNX Neutrino permits multiple sessions to be run concurrently by means of *virtual consoles*. These virtual consoles are usually named **/dev/con1**, **/dev/con2**, etc.

When the system starts `devc-con` or `devc-con-hid`, it can specify how many virtual consoles to enable by specifying the `-n`. The maximum number of virtual consoles is nine.

You can configure your system to initially launch a program on various consoles, or you can configure the `tinit` program to automatically launch (and relaunch) different programs on different consoles, based on the contents of the **/etc/config/ttys** file. By default, `tinit` launches a `login` command on the consoles.

If you increase the number of consoles on your machine, make sure you edit **/etc/config/ttys** so that `tinit` will know what to start on the additional consoles.

Each virtual console can be running a different foreground application that uses the entire screen. The keyboard is attached to the virtual console that's currently visible. You can switch from one virtual console to another, and thus from one application to another, by entering these keychords:

If you want to go to the:	Press:
Next active console	**Ctrl–Alt–Enter** or **Ctrl–Alt–+**
Previous active console	**Ctrl–Alt––**

Use the **+** (plus) and **–** (minus) keys in the numeric keypad for these keychords.

You can also jump to a specific console by typing **Ctrl–Alt–n**, where *n* is a digit that represents the console number of the virtual console. For instance, to go to **/dev/con2** (if available), press **Ctrl–Alt–2**.

When you terminate the session by typing `logout` or `exit`, or by pressing **Ctrl–D**, the console is once again idle. It doesn't appear when you use any of the cyclical console-switching keychords. The exception is console 1, where the system usually restarts `login`.

For more information about the console, see `devc-con` and `devc-con-hid` in the *Utilities Reference*, and “Console devices” in the Character I/O chapter of the *System Architecture* guide.

Shell

After the character-device driver processes what you type, the command line is passed to a command interpreter or shell.

The default shell is `sh`, which under QNX Neutrino is a link to the Korn shell, `ksh`. There are other shells available, including small ones that are suitable for situations with limited memory:

esh

Embedded shell.

fesh

Fat embedded shell; similar to `esh`, but with additional builtin commands.

uesh

Micro-embedded shell with a subset of `esh`'s functionality.

Here's a brief comparison of the features that the shells support:

Feature	uesh	esh	fesh	ksh
Interactive mode	Yes	Yes	Yes	Yes
Script files	Yes	Yes	Yes	Yes
Redirection	Yes	Yes	Yes	Yes
Pipes	—	Yes	Yes	Yes
Aliases	—	Yes	Yes	Yes
Filename expansion	—	Yes	Yes	Yes
Parameter substitution	—	Yes	Yes	Yes
Compound commands	—	—	—	Yes
Command or arithmetic substitution	—	—	—	Yes
Command and filename completion	—	—	—	Yes
Tilde expansion	—	—	—	Yes
Brace expansion	—	—	—	Yes
Coprocesses	—	—	—	Yes
Functions	—	—	—	Yes
`emacs` interactive command-line editing	—	—	—	Yes

Feature	`uesh`	`esh`	`fesh`	`ksh`
Job control	—	—	—	Yes

The small shells have fewer builtin commands than `ksh` has. For more information about these shells, see the *Utilities Reference*.

In general terms, the shell breaks the command line into tokens, parses them, and invokes the program or programs that you asked for. The specific details depend on the shell that you're using; this section describes what `ksh` does.

As you type, the Korn shell immediately processes the keys that you use to *edit the command line*, including *completing commands and filenames*. When you press **Enter**, the shell processes the command line:

1. The shell breaks the command line into tokens that are delimited by whitespace or by the special characters that the shell processes.
2. As it forms words, the shell builds commands:
    - *simple commands*, usually programs that you want to run (e.g., `less my_file`)
    - *compound commands*, including *reserved words*, grouping constructs, and function definitions

   You can also specify *multiple commands* on the command line.
3. The shell processes *aliases* recursively.
4. The shell does any required *substitutions*, including parameters, commands, and filenames.
5. The shell does any *redirection*.
6. The shell matches the remaining commands, in this order: special builtins; functions; regular builtins; executables.

To override the order in which the shell processes the command line, you use *quoting* to change the meaning of the special characters.

The sections that follow give the briefest descriptions of these steps—`ksh` is a very powerful command interpreter! For more details, see its entry in the *Utilities Reference*.

Editing the command line

The Korn shell supports `emacs`-style commands that let you edit the command line.

If you want to:	Press:
Move to the beginning of the line	**Ctrl–A**
Move to the end of the line	**Ctrl–E**
Move to the end of the current word	**EscF**
Move to the beginning of the current word	**EscB**
Delete the character at the cursor	**Ctrl–D**

If you want to:	Press:
Delete the character before the cursor	**Ctrl–H**
Delete from the cursor to the end of the current word	**EscD**
Delete from the cursor to the end of the line	**Ctrl–K**
Paste text	**Ctrl–Y**

As in `emacs`, commands that involve the **Ctrl** key are keychords; for commands that involve **Esc**, press and release each key in sequence. For more information, see "`emacs` interactive input-line editing" in the documentation for `ksh`.

In order to process these commands, `ksh` uses the character device in raw mode, but emulates all of the driver's processing of the keys. Other shells, such as `esh`, use the character device in canonical (edited input) mode.

Command and filename completion

You can reduce the amount of typing you have to do by using command completion and filename completion.

To do this, type part of the command's or file's name, and then press **Esc** *twice* (i.e., **Esc Esc**) or **Tab** *once*. The shell fills as much of the name as it can; you can then type the rest of the name—or type more of it, and then press **Esc Esc** or **Tab** again.

For example, suppose your system has executables called `my_magnificent_app` and `my_wonderful_app`:

- If you type `my_` followed by **Esc Esc** or **Tab**, the shell can't complete the command name because what you've typed isn't enough to distinguish between the possibilities.
- If you type `my_w` followed by **Esc Esc** or **Tab**, the system completes the command name, `my_wonderful_app`.

If you haven't typed enough to uniquely identify the command or file, you can press **Esc =** to get a list of the possible completions.

You can control which keys the shell uses for completing names by setting the shell's `complete` key binding. For example, the command that lets you use the **Tab** key is as follows:

```
bind '^I'=complete
```

You can use `bind` on the command line or in the `ksh` profile. For more information about the `bind` command and the key bindings, see "`emacs` interactive input-line editing" in the documentation for `ksh` in the *Utilities Reference*; for information about the profiles for `ksh`, see also "*Configuring your shell*" in Configuring Your Environment.

Reserved words

The Korn shell recognizes these reserved words and symbols:

```
case    else    function    then    !
do      esac    if          time    [[
done    fi      in          until   {
elif    for     select      while   }
```

and uses them to build compound commands. For example, you can execute commands in a loop:

```
for i in *.c; do cp $i $i.bak; done
```

Entering multiple commands

You can enter more than one command at a time by separating your commands with a semicolon (`;`).

For example, if you want to determine your current working directory, invoke `pwd`. If you want to see what the directory contains, use `ls`. You could combine the two commands as follows:

```
pwd; ls
```

As described in "*Pipes*," you can also use pipes (`|`) to connect commands on the command line.

Aliases

You can define an *alias* in the shell to create new commands or to specify your favorite options.

For example, the `-F` option to the `ls` command displays certain characters at the end of the names to indicate that the file is executable, a link, a directory, and so on. If you always want `ls` to use this option, create an alias:

```
alias ls='ls -F'
```

If you ever want to invoke the generic `ls` command, specify the path to the executable, or put a backslash (\) in front of the command (e.g., `\ls`).

Aliases are expanded in place, so you can't put an argument into the middle of the expanded form; if you want to do that, use a shell function instead. For example, if you want a version of the `cd` command that tells you where you end up in, type something like the following in `ksh`:

```
function my_cd
{
  cd $1
  pwd
}
```

For more information, see "Functions" in the entry for `ksh` in the *Utilities Reference*.

For information on adding an alias or shell function to your profile so that it's always in effect, see "*`ksh`'s startup file*" in Configuring Your Environment.

Substitutions

The shell lets you use a shorthand notation to include the values of certain things in the command line.

The shell does the following substitutions, in this order:

1. directories—tilde expansion
2. parameters
3. commands
4. arithmetical expressions
5. braces
6. filename generation

Let's look at these in more detail:

Directories—tilde expansion

The shell interprets the tilde character (~) as a reference to a user's home directory. The characters between the tilde and the next slash (if any) are interpreted as the name of a user.

For example, **~mary/some_file** refers to **some_file** in the home directory of the user named **mary**. If you don't specify a user name, it's assumed to be yours, so **~/some_file** refers to **some_file** in your home directory.

Your home directory is defined in your entry in the password database; see the description of */etc/passwd* in Managing User Accounts.

Parameters

To include the value of a parameter on the command line, put a dollar sign ($) before the parameter's name. For example, to display the value of your *PATH* environment variable, type:

```
echo $PATH
```

Commands

Sometimes, you might want to execute a command and use the results of the command in another command. You can do it like this:

```
$(command)
```

or with the older form, using backquotes:

```
`command`
```

For example, to search all of your C files for a given string, type:

```
grep string $(find . -name "*.c")
```

The `find` command searches the given directory (**.** in this case) and any directories under it for files whose names end in **.c**. The command substitution causes `grep` to search for the given string in the files that `find` produces.

Arithmetical expressions

To specify an arithmetical expression in a command line, specify it as follows:

```
$(( expression ))
```

For example:

```
echo $((5 * 7))
```

You're restricted to integer arithmetic.

Braces

You can use braces to add a prefix, a suffix, or both to a set of strings. Do this by specifying:

```
[prefix]{str1,...,strN}[suffix]
```

where commas (`,`) separate the strings. For example, `my_file.{c,o}` expands to `my_file.c my_file.o`.

Filename generation

Instead of using a command to work on just one file or directory, you can use wildcard characters to operate on many.

If you want to:	Use this wildcard:
Match zero or more characters	`*`
Match any single character	`?`
Match any characters (or range of characters separated by a hyphen) specified within the brackets	`[]`
Exclude characters specified within brackets	`!`

Hidden files, whose names start with a dot (e.g., **.profile**), aren't matched unless you specify the dot. For example, `*` doesn't match **.profile**, but `.*` does.

The following examples show you how you can use wildcards with the `cp` utility to copy groups of files to a directory named **/tmp**:

If you enter:	The `cp` utility copies:
`cp f* /tmp`	All files starting with `f` (e.g., **frd.c**, **flnt**)
`cp fred? /tmp`	All files beginning with `fred` and ending with one other character (e.g., **freda**, **fred3**)
`cp fred[123] /tmp`	All files beginning with `fred` and ending with `1`, `2`, or `3` (i.e., **fred1**, **fred2**, and **fred3**)
`cp *.[ch] /tmp`	All files ending with `.c` or `.h` (e.g., **frd.c**, **barn.h**)
`cp *.[!o] /tmp`	All files that don't end with `.o`
`cp *.{html,tex}`	All files that end with `.html` or `.tex`

Redirecting input and output

You can override the behavior of commands that read from, or write to, `stdin`, `stdout`, and `stderr`.

Most commands:

- read their input from the standard input stream (*stdin*, or file descriptor 0), which is normally assigned to your keyboard
- write their output to the standard output file (*stdout*, or fd 1), which is normally assigned to your display screen
- write any error messages to the standard error stream (*stderr*, or fd 2), which is also normally assigned to the screen

Sometimes you want to override this behavior.

If you want a process to:	Use this symbol:
Read from a file, or another device (input redirection)	<
Write *stdout* to a file (output redirection)	>
Write *stdout* to a file, appending to the file's contents (output append)	>>

For example, the `ls` command lists the files in a directory. If you want to redirect to output of `ls` to a file called **filelist**, enter:

```
ls > filelist
```

You can specify a file descriptor for the above redirections. For example, if you don't want to display any error messages, redirect *stderr* to **dev/null** (a special file, also known as the bit bucket, that swallows any data written to it and returns end-of-file when read from):

```
my_command 2> /dev/null
```

For more information, see "Input/output redirection" in the entry for `ksh` in the *Utilities Reference*.

Pipes

You can use a pipe (`|`) to build complex commands from smaller ones.

For example:

```
grep 'some term' *.html | sort -u | wc -l
```

Programs such as `grep`, `sort`, and `wc` (a utility that counts characters, words, and lines) that read from standard input and write to standard output are called *filters*.

Quoting special characters

Certain characters may have special meaning to the shell, depending on their context. If you want a command line to include any of the special characters that the shell processes, then you may have to *quote* these characters to force the shell to treat them as simple characters.

You *must* quote the following characters to avoid their special interpretation:

`| $ ( " ) & ` ; \ '` **Tab Newline Space**

You *might* need to quote the following characters, depending on their context within a shell command:

`* ? [ # ~ = %`

In order to quote:	You can:
A single character	Precede the character with a single backslash (`\`) character
All special characters within a string of characters	Enclose the whole string in *single* quotes
All special characters within a string, except for `$`, `` ` ``, and `\`	Enclose the whole string in *double* quotes

For example, these commands search for all occurrences of the string "realtime OS" in the **chapter1.html** file:

```
grep realtime\ OS chapter1.html
grep 'realtime OS' chapter1.html
grep "realtime OS" chapter1.html
```

However, note that:

```
grep realtime OS chapter1.html
```

doesn't do what you might expect, as it attempts to find the string “realtime” in the files named **OS** and **chapter1.html**.

Depending on the complexity of a command, you might have to nest the quoting. For example:

```
find -name "*.html" | xargs grep -l '"realtime.*OS"' | less
```

This command lists all the HTML files that contain a string consisting of `realtime`, followed by any characters, followed by `OS`. The command line uses `find` to locate all of the files with an extension of `html` and passes the list of files to the `xargs` command, which executes the given `grep` command on each file in turn. All of the output from `xargs` is then passed to `less`, which displays the output, one screenful at a time.

This command uses quoting in various ways to control when the special characters are processed, and by which process:

- If you don't put quotes around the `*.html`, the shell interprets the `*`, and passes to `find` the list of files in the current directory with an extension of `html`. If you quote the `*.html`, the shell passes the string as-is to `find`, which then uses it to match all of the files in this directory and below in the filesystem hierarchy with that extension.
- In a similar way, if you don't quote the `realtime.*OS` string at all, the shell generates a list of files that match the pattern. Quoting it once (`"realtime.*OS"`) works for a single invocation of `grep`, but this example has the added complexity of the `xargs` command.
- The `xargs` command takes a command line as its argument, and the shell interprets this command line for each item that's passed to `xargs`. If you don't want the `realtime.*OS` string to be interpreted by the shell at all, you need to put nested quotes around the pattern that you want to pass to `grep`:

  ```
  xargs grep -l '"realtime.*OS"'
  ```

- The quoting also indicates when you want to execute the `less` command. As given, the shell passes the output from *all* of the invocations of `xargs` to `less`. In contrast, this command:

  ```
  find -name "*.html" | xargs 'grep -l "realtime.*OS" | less'
  ```

  passes the command:

  ```
  grep -l "realtime.*OS" | less
  ```

  to `xargs`, which will have quite different results—if it works at all.

For more information, see “Quoting” in the entry for `ksh` in the *Utilities Reference*.

History: recalling commands

The shell lets you recall commands that you've previously entered; use the up and down arrows to move through the history buffer. You can edit the command, if you wish, and then press **Enter** to reexecute it.

The shell also includes a builtin `fc` command that you can use to display and edit previous commands, as well as an `r` alias to `fc` that reexecutes a previous command. For example:

```
r string
```

reexecutes the last command that starts with the given string.

Shell scripts

You can enter shell commands into a text file, called a *shell script*, and then invoke the commands in batch mode by executing (or shelling) the file. For more information, see the Writing Shell Scripts chapter in this guide.

Utilities

Once the shell has processed all of its special characters, what remains typically consists of commands and the arguments to them. Most commands correspond to executable files somewhere on your system, although some—such as `cd`—are built into the shell.

Give us the tools, and we will finish the job.

—Sir Winston Churchill

It's possible for you to have more than one executable file with the same name on your system. The shell uses the *PATH* environment variable to determine which version to use.

The value of *PATH* is a list of directories, separated by colons (**:**), in the order in which you want the shell to search for executables. To see the value of your *PATH*, type:

```
echo $PATH
```

CAUTION: You can put your current directory (**.**) in your *PATH*, but it can leave you vulnerable to "Trojan horse" programs. For example, if **.** is at the beginning of your *PATH*, the shell looks in the current directory first when trying to find a program. A malicious user could leave a program called `ls` in a directory as a trap for you to fall into.

If you want to have your current directory in your *PATH*, make sure that you put it *after* the directories that hold the common utilities.

For information about setting your *PATH*, see "*Environment variables*" in Configuring Your Environment.

If you want to know which version of a command the shell will choose, use the `which` command. For example:

```
$ which ls
/bin/ls
```

You can use command-line options to get more information:

```
$ which -laf ls
-rwxrwxr-x  1 root      root         19272 May 03  2002 /bin/ls
```

If you try this for a command that's built into the shell, `which` can't find it:

```
$ which cd
which: no cd in /bin:/usr/bin:/opt/bin
```

The `whence` command displays what the command means to the shell, including any aliases in effect. For example, if you've created an alias for `ls`, the output might be:

```
$ whence ls
'ls -F'
```

Understanding command syntax

Whenever you look up a command in the *Utilities Reference*, you'll see a syntax statement that summarizes how you can use the command.

For most commands, this statement consists of the following components:

command_name

The name of the command to be executed. This may be the name of an executable program, such as a utility, or it may be the name of a command built into the shell.

options

The specific behavior that you want to invoke for the command. Options typically consist of an alphanumeric character preceded by a hyphen (e.g., `-c`). Some options take an argument (e.g., `-n` *number*). If you specify an option that takes an argument, you must include its argument as well.

operands

Data the command requires (e.g., a filename). If a command lets you enter multiple operands, they're usually processed in the order you list them. Unlike options, operands aren't preceded by a hyphen (e.g., `less my_file`).

The entries in the *Utilities Reference* use some special symbols to express the command syntax:

...

You can specify one or more instances of the previous element. For example, in the `less` utility syntax, the ellipsis after the operand *file* indicates that you can specify more than one file on the command line:

```
less myfile1 myfile2
```

[ ]

The enclosed item is optional.

|

You can use only one of the items (e.g., `-a|-f`).

You don't actually type these symbols when you invoke the command. For instance, the syntax description for `less` is given as follows:

```
less [-[+]aBcCdeEfimMnNqQrsSuUw] [-b n] [-x n]
     [-[z] n] [-h n] [-j n] [-p pattern]
     [-y n] [-[oO] logfile] [-t tag]
     [-T tagsfile] [+ cmd] [file...]
```

You can combine multiple options that don't take an argument. The `-aBcCdeEfimMnNqQrsSuUw` notation is shorthand for `-a -B -c -C -d` and so on.

If an argument to a command starts with a hyphen, you can signal the end of the options by using a double hyphen:

```
ls -l -- -my_file
```

For more information, see Utility Conventions in the *Utilities Reference*.

Displaying online usage messages

You can look up detailed usage descriptions or you can display brief summaries of the syntax and options of commands and utilities.

If you want a detailed description of a utility, see the *Utilities Reference*. But if you just want a quick reminder of the syntax and options, you can display the utility's online usage message by invoking the `use` command (it's similar to `man` in Unix and Linux). For example, to display the message for `more`, type:

```
use more
```

If you request usage for a command, and the command either doesn't have an executable in the current path or doesn't contain usage message records, `use` displays an error message. For more information, see `use` in the *Utilities Reference*—or simply type `use use`.

Executing commands on another node or tty

If the machines on your network are running Qnet (see Using Qnet for Transparent Distributed Processing), you can execute commands on another machine.

This is known as *remote execution*. For example:

```
on -n /net/dasher date
```

where **/net/dasher** is the name of the node that you want to run the command on.

When you invoke a command on another node, the command's standard input, standard output, and standard error output are displayed on your console screen (or terminal) unless you explicitly redirect them to another device.

To run a command on a specific tty, use the `-t` option, specifying the terminal name. For example:

```
on -t con3 login root
```

For more information, see the `on` command in the *Utilities Reference*.

Priorities

By default, when you start a utility or other program, it runs at the same priority as its parent. (Actually, priorities aren't associated with a process, but with the process's threads.) You can determine the priority of a process's threads by looking at the output of the `pidin` (Process ID INformation) command.

If you want to run something at a specific priority, use `on`, specifying the `-p` option. If you want to specify a relative priority, use the `nice` command.

Basic commands

Here are some QNX Neutrino commands that you'll frequently use:

If you want to:	Use:
Determine your current directory	`pwd` (builtin `ksh` command)
Change directory	`cd` (builtin `ksh` command)
List the contents of a directory	`ls`
Rename (move) files and directories	`mv`
Delete (remove) files	`rm`
Copy files and file hierarchies	`cp` or `pax`
Create directories	`mkdir`
Remove directories	`rmdir`
Determine how much free space you have on a filesystem	`df`
Concatenate and display files	`cat`
Display output on a page-by-page basis	`less` or `more`
Find files based on search criteria	`find`
Change a file's permissions/attributes	`chmod`
Create hard and symbolic links	`ln`
Create a "tape archive"	`tar` or `pax`
Extract files from a **.tar** file	`tar`
Extract files from a **.tar.gz** or **.tgz** file	`gunzip` *filename* `| pax -r` or `tar -xzf` *filename*

For more information about these and other commands, see the *Utilities Reference*.

Troubleshooting

Here are some common problems you might encounter while working on the command line.

Why can't I run my program called* test*?

The shell has a builtin command called `test`. When the shell parses the command line, it matches any builtin commands before it looks for executable files.

You have two choices: rename your program, or specify the path to it (e.g., **./test**).

Why do I get a "not found" message when I try to run my program?

The program is likely in a directory that isn't listed in your *PATH*. In particular, your current directory isn't in your *PATH* for security reasons.

Either add the executable's directory to your *PATH* or specify the path to the command (e.g., **./my_program**). For more information, see "*Utilities*," earlier in this chapter.

When I list a directory, I don't see files that start with a dot.

Files whose names start with a dot (`.`) are called *hidden files*. To list them, use the `-a` option to `ls`.

Why am I getting a "No such file or directory" message?

The shell can't find the file or directory that you specified. Here are some things to check:

- Have you typed the name correctly? In QNX Neutrino, the names of files and directories *are* case-sensitive.
- Does the name contain spaces or other special characters?

  If you have a file called **my file** and you don't escape the meaning of the space, the shell uses the space when breaking the command line into tokens, so the command looks for one file called **my** and another called **file**.

  Use quoting to escape the meaning of the special characters (e.g., `less "my file"` or `less my\ file`). For information about the other characters that you need to quote, see "*Quoting special characters*."

How do I work with a file whose name starts with a hyphen?

QNX Neutrino utilities use the hyphen (`-`) to denote an option (e.g., `head -n 10 some_file`). If you create a file whose name starts with a hyphen, and you pass that filename as an argument to a utility, the utility parses the filename as one or more options.

Most utilities recognize a double hyphen (`--`) to mean "end of options." Put this before your filename:

```
head -- -my_file
```

For more information, see the Utility Conventions chapter in the *Utilities Reference*.

Why do I get a "Unrecognized TERM type" message when I start programs such as `vi`?

Either your *TERM* environment variable isn't set correctly, or there isn't an entry for your terminal type in **/usr/lib/terminfo/** (or possibly **/etc/termcap**); see "*Terminal support*," earlier in this chapter.

Chapter 5
Working with Files

In a QNX Neutrino system, almost everything is a file; devices, data, and even services are all typically represented as files. This lets you work with local and remote resources easily from the command line, or through any program that works with files.

This chapter concentrates on working with files in the Power-Safe filesystem (`fs-qnx6.so`), which is the default under the QNX Neutrino RTOS. For more information, see the *Working with Filesystems* chapter in this guide.

Types of files

QNX Neutrino supports various types of files. The file type gives you a hint about what type of data the file contains and how you should expect the file to behave.

The `ls -l` command uses the character shown in parentheses below to identify the file type:

Regular (`-`)

A file that contains user data, such as C code, HTML, and data. For example, **/home/fred/myprog.c**.

Directory (`d`)

Conceptually, a directory is something that contains files and other directories. For example, **/home/fred**.

Symbolic link (`l`)

An additional name for a file or directory. For example, **/usr/bin/more** is a symbolic link to `/usr/bin/less`. For more information, see "*Links and inodes* " in Working with Filesystems.

Named special (`n`)

A special-purpose file that doesn't fit the behavior or content type of any other type of file. For example, **/proc/dumper** represents the hook for generating core dumps when an application crashes, and **/dev/random** represents a source of random numbers.

Character special files (`c`)

Entries that represent a character device. For example, **/dev/ser1** represents a serial port.

FIFO special files (`p`)

Persistent named pipes through which two programs communicate. For example, **PipeA**.

Block special files (`b`)

Entries that represent a block device, such as a disk. For example, **/dev/hd0** represents the raw block data of your primary disk drive.

Socket files (`s`)

Entries that represent a communications socket, especially a Unix-domain socket. For more information, see *socket()* and the Unix protocol in the QNX Neutrino *C Library Reference*.

Some files are persistent across system reboots, such as most files in a disk filesystem. Other files may exist only as long as the program responsible for them is running. Examples of these include shared memory objects, objects in the **/proc** filesystem, and temporary files on disk that are still being accessed even though the links to the files (their filenames) have been removed.

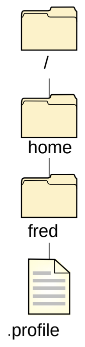
Filenames and pathnames

To access any file or directory, you must specify a *pathname*, a symbolic name that tells a program where to find a file within the directory hierarchy based at root (**/**).

A typical QNX Neutrino pathname looks like this:

/home/fred/.profile

In this example, **.profile** is found in the **fred** directory, which in turn resides in the **home** directory, which is found in **/**, the *root directory*:



Like Linux and other Unix-like operating systems, QNX Neutrino pathname components are separated by a forward slash (/). This is unlike Microsoft operating systems, which use a backslash (\).

To explore the files and directories on your system, use the `ls` utility. This is the equivalent of `dir` in MS-DOS. For more information, see "*Basic commands*" in Using the Command Line, or `ls` in the *Utilities Reference*.

Absolute and relative pathnames

There are two types of pathname:

Absolute paths

Pathnames that begin with a slash specify locations that are relative to the root of the pathname space (**/**). For example, **/home/fred/my_apps/favs**.

Relative paths

Pathnames that don't begin with / specify locations relative to your current working directory.

For example, if your current directory is **/home/fred**, a relative path of **my_apps/favs** is the same as an absolute path of **/home/fred/my_apps/favs**.

You can't tell by looking at a pathname whether the path points to a regular file, a directory, a symbolic link, or some other file type. To determine the type of a file, use `file` or `ls -ld`.

The one exception to this is a pathname that ends with /, which always indicates a directory. If you use the `-F` option to `ls`, the utility displays a slash at the end of a directory name.

Dot and dot-dot directories

Most directories contain two special links, **.** (dot) and **..** (dot dot).

. ("dot")

The current directory.

.. ("dot dot")

The directory that this directory appears in.

So, for example, you could list the contents of the directory above your current working directory by typing:

```
ls ..
```

If your current directory is **/home/fred/my_apps/favs**, you could list the contents of the root directory by typing:

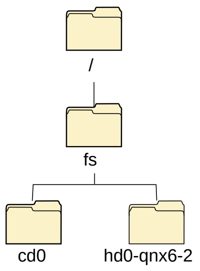
```
ls ../../../..
```

but the absolute path (**/**) is much shorter, and you don't have to figure out how many "dot dots" you need.

No drive letters

Unlike Microsoft Windows, which represents drives as letters that precede pathnames (e.g., **C:\**), QNX Neutrino represents disk drives as regular directories within the pathname space. Directories that access another filesystem, such as one on a second hard disk partition, are called *mountpoints*.

Usually the primary disk-based filesystem is mounted at **/** (the root of the pathname space). A full QNX Neutrino installation mounts all additional disk filesystems automatically under the **/fs** directory. For example:



So, while in a DOS-based system a second partition on your hard drive might be accessed as **D:\**, in a QNX Neutrino system you might access the second Power-Safe filesystem partition on the first hard drive as **/fs/hd0-qnx6-2**.

For more information on where to find things in a typical QNX Neutrino pathname space, see "*Where everything is stored*," later in this chapter. To learn more about mounting filesystems, see *Working with Filesystems*.

Pathnames that begin with a dot

When you list the contents of a directory, the `ls` utility usually hides files and directories whose names begin with a period. Programs precede configuration files and directories with a period to hide them from view. The files (not surprisingly) are called *hidden files*.

Other than the special treatment by `ls` and some other programs, nothing else is special about hidden files. Use `ls -a` to list all files, including any hidden ones.

Extensions

Filename extensions (`.`*something* at the end of a filename) tell programs and users what type of data a file contains.

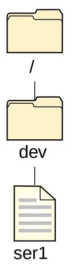
In QNX Neutrino filesystems, extensions are just an ordinary part of the filename and can be any length, as long as the total filename size stays within the 505-byte filename length limit.

Most of the time, file extensions are simply naming conventions, but some utilities base their behavior on the extension. See "*Filename extensions*" for a list of some of the common extensions used in a QNX Neutrino system.

Pathname-space mapping

You may have noticed that we've talked about files and directories *appearing in* their parent directories, rather than just saying that the parent directories contain these files. This is because in QNX Neutrino, the pathname space is virtual, dictated not just by the filesystem that resides on media mounted at root, but rather by the paths and pathname aliases registered by the process manager.

For example, let's take a small portion of the pathname space:



In a typical disk-based QNX Neutrino system, the directory **/** maps to the root of a filesystem on a physical hard drive partition. This filesystem on disk doesn't actually contain a **/dev** directory, which exists virtually, adopted via the process manager. In turn, the filename **ser1** doesn't exist on a disk filesystem either; it has been adopted by the serial port driver.

This capability allows virtual *directory unions* or *unioned filesystems* to be created. This happens when multiple resource managers adopt files that lie in a common directory within the pathname space.

Which resource manager looks after which file or directory depends on the length of the matched pathname and order in which the underlying filesystems are mounted, as described in "Pathname Management" in the Process Manager chapter of the *System Architecture* guide.

The more filesystems involved in the union, the more complicated things become. In the interests of creating a maintainable system, we suggest that you create directory unions as rarely as possible.

For a more detailed example, see “Unioned filesystems” in the Resource Managers chapter of *Getting Started with QNX Neutrino*.

Filename rules

QNX Neutrino supports a variety of filesystems, each of which has different capabilities and rules for valid filenames.

For information about filesystem capabilities, see the *Working with Filesystems* chapter; for filesystem limits, see the *Understanding System Limits* chapter.

The maximum length of a filename depends on the type of filesystem. Individual bytes within the filename may have any value *except* the following (all values are in hexadecimal):

- `0x00` through `0x1F` (all control characters)
- `0x2F` (/)
- `0x7F` (rubout)
- `0xFF`

If you're using UTF-8 representations of Unicode characters to represent international characters, the limit on the filename length will be lower, depending on your use of characters in the extended range.

In the Power-Safe filesystem, you can use international characters in filenames by using the UTF-8 encoding of Unicode characters. Filenames containing UTF-8 characters are generally illegible when viewed from the command line.

You can also use the ISO-Latin1 supplemental and PC character sets for international characters; however, the appearance of these 8-bit characters depends on the display settings of your terminal, and might not appear as you expect in other operating systems that access the files via a network.

Most other operating systems, including Microsoft Windows, support UTF-8/Unicode characters. Filenames from older versions of Microsoft Windows may be encoded using 8-bit characters with various language codepage in effect. The DOS filesystem in QNX Neutrino can translate these filenames to UTF-8 representations, but you need to tell the filesystem which codepage to use via a command-line option. For more information see **fs-dos.so** in the *Utilities Reference*.

All our disk filesystems—i.e., **fs-dos.so**, **fs-ext2.so**, the Power-safe filesystem (**fs-qnx6.so**), and **fs-udf.so**—use UTF-8 encoding for presentation of their filenames; attempts to specify a filename not using UTF-8 encoding will fail (with an error of EILSEQ) on these filesystems.

Where everything is stored

The default QNX Neutrino filesystem generally follows the Filesystem Hierarchy Standard, but we don't claim to be compliant or compatible with it.

This standard describes where files and directories should or must be placed in Unix-style operating systems. For more information, see *http://www.pathname.com*

The QNX Neutrino pathname space is extremely flexible. Your system may be configured differently.

/

The **/** directory is the root of the pathname space. Usually your primary hard disk or flash filesystem is mounted here.

On a Power-Safe (**fs-qnx6.so**) filesystem, this directory includes the following:

/.boot

A *directory* that contains the OS images that the secondary boot loader can load on bootup.

as well as the directories described in the sections that follow.

/bin

The **/bin** directory contains binaries of essential utilities, such as `chmod`, `ls`, and `ksh`.

To display basic utility syntax, type `use` *utilityname* from the command line. For more information, see `use` in the *Utilities Reference*.

/dev

The **/dev** directory belongs to the process manager and contains device files, possibly including:

/dev/cd*n*

CD-ROM block devices; see `devb-*` in the *Utilities Reference* for driver information.

/dev/con*n*

Text mode console TTY device; see `devc-con` in the *Utilities Reference*.

/dev/console

The device that's used for diagnostic log messages; on a full x86 system, this is a write-only device managed by the system logger, `slogger2`. Buildfiles for embedded systems may configure a link from this path to another device, such as a serial port. See `slogger2` in the *Utilities Reference*.

/dev/hd*n*

Hard disk block devices; data representing an entire drive, spanning all partitions; see `devb-*` in the *Utilities Reference*.

/dev/hd*n*t*n*

Hard disk partition block devices; the data in these devices is a subset of that represented by the corresponding **hd***n* file; see `devb-*` in the *Utilities Reference*.

/dev/mem

A device that represents all physical memory.

/dev/mq and /dev/mqueue

A pathname space where entries for message queues appear; for more information, see `mq` and `mqueue` in the *Utilities Reference*.

/dev/null

A "bit bucket" that you can direct data to. The data is discarded.

/dev/pci

Adopted by the PCI server on the machine, this device lets programs communicate with the PCI server. See `pci-*` in the *Utilities Reference*.

/dev/pipe

Adopted by the `pipe` manager. The presence of this file tells other programs (such as a startup script built into an OS image) that the Pipe manager is successfully running.

/dev/pty[p-zP-T][0-9a-f]

The control side of a pseudo-terminal device pair. Pseudo-ttys are named with a letter (p–z or P–T) followed by a hexadecimal digit, making it possible to have up to 256 devices. See `devc-pty` in the *Utilities Reference*.

/dev/random

Read from this device to obtain random data; see `random` in the *Utilities Reference*.

/dev/sem

A pathname space where entries for named semaphores appear.

/dev/ser*n*

Serial ports. See `stty` for configuration, and `devc-ser*` for driver details in the *Utilities Reference*.

/dev/shmem/

Contains files representing shared memory regions on the system (also sometimes used for generic memory-mapped files). For more information, see the description of the *RAM "filesystem"* in Working with Filesystems.

/dev/socket/

This directory is owned and managed through the TCP/IP stack, which is included in `io-pkt*`. This directory contains pathnames through which applications interact with the stack. For more information, see the *TCP/IP Networking* chapter in this guide.

/dev/text

This file is managed by `procnto`. Text written to this device is output through debug output routines encoded in the startup code for your system, so the actual result varies from board to board. On a standard PC, the default is to write to the PC console. For more information, see `startup-*` in the *Utilities Reference*.

/dev/tty

A virtual device owned by the process manager (`procnto`) that resolves to the controlling terminal device associated with the session of any process that opens the file. This is useful for programs that may have closed their standard input, standard output, or standard error, and later wish to write to the terminal device.

/dev/tty[p-zP-T][0-9a-f]

The slave side of the corresponding **/dev/pty[p-zP-T][0-9a-f]** file. The program being controlled typically uses one of these files for its standard input, standard output, and standard error.

/dev/zero

Supplies an endless stream of bytes having a value of zero.

/etc

The **/etc** directory contains host-specific system files and programs used for administration and configuration, including:

/etc/bootptab

Network boot protocol server configuration file. See **/etc/bootptab** in the *Utilities Reference*.

/etc/config/

A directory that contains system-configuration files, such as the **ttys** file that `tinit` uses to configure terminal devices.

/etc/default/

A directory that contains default configuration files, primarily for TCP/IP facilities.

/etc/dhcpd.conf

Dynamic Host Configuration Protocol configuration; see **/etc/dhcpd.conf** in the *Utilities Reference*.

/etc/ftpd.conf

Configuration options for `ftpd` that apply once you've authenticated your connection. See ../../com.qnx.doc.neutrino.utilities/topic/f/ftpd.conf.html in the *Utilities Reference*.

/etc/ftpusers

Defines users who may access the machine via the File Transfer Protocol. See **/etc/ftpusers** in the *Utilities Reference*.

/etc/group

User account group definitions; see *Managing User Accounts*.

/etc/hosts

Network hostname lookup database; see also **/etc/nsswitch.conf** and **/etc/resolv.conf**, below. See **/etc/hosts** in the *Utilities Reference*.

/etc/inetd.conf

Internet super-server configuration file that defines Internet services that `inetd` starts and stops dynamically as needed.

The descriptions in the default version of this file are commented out; uncomment the ones that you want to use. See **/etc/inetd.conf** in the *Utilities Reference*.

/etc/motd

Contains an ASCII message of the day that may be displayed when users log in, as long as **/etc/profile** is configured to display it.

The default **/etc/profile** displays this file only if the **/etc/motd** file is more recent than the time you last logged in to the system, as determined by the time your ***$HOME*/.lastlogin** file was last modified. For more information, see the description of */etc/profile* in Configuring Your Environment.

/etc/networks

Network name database file. For more information, see **/etc/networks** in the *Utilities Reference*.

/etc/nsswitch.conf

Name-service switch configuration file. For more information, see **/etc/nsswitch.conf** in the *Utilities Reference*.

/etc/opasswd

Backup of **/etc/passwd** file before its last change via the `passwd` utility. See the *Managing User Accounts* chapter.

/etc/oshadow

Backup of **/etc/shadow** file before its last change via the `passwd` utility. *Managing User Accounts*.

/etc/passwd

This file defines login accounts. See the chapter *Logging In, Logging Out, and Shutting Down*, as well as *Managing User Accounts* for more details; also see `passwd`, `login` in the *Utilities Reference*.

/etc/profile

The startup profile script executed by the shell when you log in; it's executed before ***$HOME*/.profile**. See *Configuring Your Environment*.

/etc/profile.d/

A directory where the default **/etc/profile** script looks for scripts to run when any user logs in. The **/etc/profile** script runs each script in this directory that matches `*.${SHELL##*/}`. For example, if the value of the *SHELL* environment variable is **/bin/sh**, the script runs the scripts that match ***.sh**.

/etc/resolv.conf

Resolver configuration file; see also **/etc/hosts**, above. See **/etc/resolv.conf** in the *Utilities Reference*.

/etc/skel/

A directory that holds the default version of **.profile**. When you add a new user to the system, this file is copied to the user's home directory. For more information, see the description of **/etc/default/passwd** in the documentation for `passwd`, and the description of *.profile* in Configuring Your Environment.

/home

The home directories of regular users are found here. The name of your home directory is often the same as your user name. You can create whatever directory structure you need inside your home directory.

/lib

This directory contains essential shared libraries that programs need in order to run (*filename***.so**), as well as static libraries used during development. See also **/usr/lib** and **/usr/local/lib**.

The **/lib** directory includes:

/lib/dll/

Contains additional shared libraries that implement OS drivers and services, such as drivers, filesystem managers, and so on. For some examples of how shared libraries are used for certain types of drivers and services, see Filesystems, Native Networking (Qnet), and TCP/IP Networking in the *System Architecture* guide. For details about specific shared objects in the **/lib/dll** directory, see their respective entries in the *Utilities Reference*.

/proc

Owned by the process manager (`procnto`), this virtual directory can give you information about processes and pathname-space configuration.

The **/proc** directory contains a subdirectory for each process; the process ID is used as the name of the directory. Each of these directories contains entries that you can (with the appropriate permission) use to access the process's address space, control its threads, and so on. Various utilities use this entry to get information about a process. For more information, see "Controlling processes via the **/proc** filesystem" in the Processes chapter of the QNX Neutrino *Programmer's Guide*, and the **/proc** Filesystem appendix of *The QNX Neutrino Cookbook*.

The **/proc** directory also includes:

/proc/boot/

The image filesystem that comprises the boot image. For more information, see OS Images in *Building Embedded Systems*.

/proc/dumper

A special entry that receives notification when a process terminates abnormally. The `dumper` utility creates this entry.

/proc/mount/

Pathname-space mountpoints.

If you list the contents of the **/proc** directory, **/proc/mount** doesn't show up, but you can list the contents of **/proc/mount**.

For more information, see "The **/proc/mount** directory" in the **/proc** Filesystem appendix of *The QNX Neutrino Cookbook*.

/proc/qnetstats

If you're using Transparent Distributed Processing (TDP), the **lsm-qnet.so** module places a **qnetstats** entry in **/proc**. If you open this name and read from it, the Qnet resource manager code responds with the current statistics for Qnet.

/proc/self/

The address space for yourself (i.e., for the process that's making the query).

/root

The **/root** directory is the home directory for the **root** user.

/sbin

This directory contains essential system binaries, including:

- drivers (e.g., `devb*`, `devc*`, `devf*`, `devu*`)
- initialization programs (e.g., `seedres`)
- configuration utilities (e.g., `ifconfig`) and repair utilities (e.g., `chkqnx6fs`, `chkdosfs`)
- managers (e.g., `io-pkt*`, `mqueue`, `pipe`)

/tmp

This directory contains temporary files. Programs are supposed to remove their temporary files after using them, but sometimes they don't, either due to poor coding or abnormal termination. You can periodically clean out extraneous temporary files when your system is idle.

/usr

The **/usr** directory is a secondary file hierarchy that contains shareable, read-only data. It might include the following:

/usr/bin/

A directory that contains most user commands. Examples include `diff`, `errno`, and `wc`.

/usr/lib/

Object files, libraries, and internal binaries that you shouldn't execute directly or in scripts. You'll link against these libraries if you write any programs.

/usr/libexec/

A directory that could contain system daemons and system utilities; in general, these are run only by other programs.

/usr/local/

A directory where the system administrator can install software locally. It's initially empty.

/usr/sbin/

Nonessential system binaries, such as `cron`, `dumper`, and `nicinfo`.

/usr/share/

Data that's independent of the architecture, such as icons, backdrops, and various `gawk` programs.

/var

The **/var** directory contains variable data files, including cache files, lock files, log files, and so on.

/var/pps

The directory where the Persistent Publish/Subscribe manager, `pps`, stores ("persists") its state on shutdown.

File ownership and permissions

Each file and directory belongs to a specific user ID and group ID, and has a set of permissions (also referred to as modes) associated with it.

You can use these utilities to control ownership and permissions:

To:	Use:
Specify the permissions for a file or directory	`chmod`
Change the owner (and optionally the group) for a file or directory	`chown`
Change the group for a file or directory	`chgrp`

For details, see the *Utilities Reference*.

You can change the permissions and ownership for a file or directory only if you're its owner or you're logged in as **root**. If you want to change both the permissions *and* the ownership, change the permissions first. Once you've assigned the ownership to another user, you can't change the permissions.

Permissions are divided into these categories:

`u`

Permissions for the user (i.e., the owner).

`g`

Permissions for the group.

`o`

Permissions for others (i.e., everyone who isn't in the group).

Each set of permissions includes:

`r`

Read permission. For a directory, this is permission to list the directory.

`w`

Write permission.

`x`

Execute permission. For a directory, this is permission to search the directory.

`s` or **`S`**

Setuid or setgid (see below).

t or T

Sticky bit (see below).

If you have read, but not search, permission for a directory, you can see the files in the directory, but you can't read or modify the contents of the files. If you have search, but not read, permission for a directory (say **dir**) and read permission on a subdirectory (say **dir/subdir**), then you can't list the contents of **dir** to see **subdir**, but if you—somehow—know that **dir/subdir** exists, you can list the contents of **dir/subdir** if you specify its path directly.

If you list your home directory (using `ls -al`), you might get output like this:

```
total 94286
drwxr-xr-x 18 barney    techies        6144 Sep 26 06:37 ./
drwxrwxr-x  3 root      root           2048 Jul 15 07:09 ../
-rw-rw-r--  1 barney    techies         320 Nov 11  2013 .kshrc
-rw-rw-r--  1 barney    techies           0 Aug 08 09:17 .lastlogin
-rw-r--r--  1 barney    techies         254 Nov 11  2013 .profile
-rw-rw-r--  1 barney    techies        3585 Jul 31  1993 123.html
-rw-rw-r--  1 barney    techies         185 Aug 08  2014 Some_file
drwx------  2 barney    techies        4096 Jul 04 11:17 bin/
-rw-------  1 barney    techies          34 Jul 05  2002 cmd.txt
drwxr-xr-x  2 barney    techies        2048 Feb 26  2014 interesting_stuff/
drwxrwxr-x  3 barney    techies        2048 Oct 17  2002 more_stuff/
drwxrwxr-x  2 barney    techies        4096 Jul 04 09:06 workspace/
```

The first column is the set of permissions. A leading `d` indicates that the item is a directory; see "*Types of files*," earlier in this chapter.

If the permissions are followed by a plus sign (+), the file or directory has an *access control list* that further specifies the permissions. For more information, see "*Access Control Lists (ACLs)*," below.

You can also use octal numbers to indicate the modes; see `chmod` in the *Utilities Reference*.

Setuid and setgid

Some programs, such as `passwd`, need to run as a specific user in order to work properly:

```
$ which -l passwd
-rwsrwxr-x  1 root      root         21544 Mar 30 23:34 /usr/bin/passwd
```

Notice that the third character in the owner's permissions is `s`. This indicates a *setuid* ("set user ID") command; when you run `passwd`, the program runs as the owner of the file (i.e., **root**). An `S` means that the setuid bit is set for the file, but the execute bit isn't set.

You might also find some *setgid* ("set group ID") commands, which run with the same group ID as the owner of the file, but not with the owner's user ID. If setgid is set on a directory, files created in the directory have the directory's group ID, not that of the file's creator. This scheme is commonly used for spool areas, such as **/usr/spool/mail**, which is setgid and owned by the **mail** group, so that programs running as the **mail** group can update things there, but the files still belong to their normal owners.

If you change the ownership of a setuid command, the setuid bit is cleared, unless you're logged in as **root**. Similarly, if you change the group of a setgid command, the setgid bit is cleared, unless you're **root**.

When running on a Windows host, `mkefs`, `mketfs`, and `mkifs` can't get the execute (`x`), setuid ("set user ID"), or setgid ("set group ID") permissions from the file. Use the `perms` attribute to specify these permissions explicitly. You might also have to use the `uid` and `gid` attributes to set the ownership correctly. To determine whether or not a utility needs to have the setuid or setgid permission set, see its entry in the *Utilities Reference*.

CAUTION: Setuid and setgid commands can cause a security problem. If you create any, make sure that only the owner can write them, and that a malicious user can't hijack them—especially if **root** owns them.

Sticky bit

The *sticky bit* is an access permission that affects the handling of executable files and directories.

- If it's set for an executable file, the kernel keeps the executable in memory for "a while" after the program ends—the exact length of time depends on what else is happening in the system. This can improve the performance if you run a program frequently.
- For a directory, it affects who can delete a file in the directory. You always need to have write permission on the directory, but if the sticky bit is set for the directory, you also need to be the owner of the file or directory or have write permission on the file.

If the third character in a set of permissions is `t` (e.g., `r-t`), the sticky bit and execute permission are both set; `T` indicates that only the sticky bit is set.

Default file permissions

Use the `umask` command to specify the mask for setting the permissions on new files.

The default mask is `002`, so any new files give read and write permission to the user (i.e., the owner of the file) and the rest of the user's group, and read permission to other users. If you want to remove read and write permissions from the other users, add this command to your **.profile**:

```
umask 006
```

If you're the system administrator, and you want this change to apply to everyone, change the `umask` setting in **/etc/profile**. For more information about profiles, see *Configuring Your Environment*.

Access Control Lists (ACLs)

Some filesystems, such as the Power-Safe (`fs-qnx6.so`) filesystem, extend file permissions with *Access Control Lists*, which are based on the withdrawn IEEE POSIX 1003.1e and 1003.2c draft standards.

With the traditional file permissions as set with `chmod`, if you want someone to have special access to a file, you have few choices:

- adding that person to the owning group
- creating a supplemental group that includes that person and the owner of the file
- loosening the permissions for “others”

Keeping track of the users in each group and can become complicated, and allowing “others” additional permissions can make your system less secure. ACLs extend file permissions, giving you finer control over who has access to what. In an ACL, the permissions are divided into these classes:

- owner class
- group class, consisting of named users, the owning group, and named groups
- others (or world) class

An access control list consists of a number of entries, each in one of the following forms (given with the constants used in code to identify the tag type):

Entry type	Tag type	Form
Owner	ACL_USER_OBJ	`user::`*permissions*
Named user (identified by name or by numerical ID)	ACL_USER	`user:`*user_identifier*`:`*permissions*
Owning group	ACL_GROUP_OBJ	`group::`*permissions*
Named group (identified by name or numerical ID)	ACL_GROUP	`group:`*group_identifier*`:`*permissions*
The upper bound on permissions for the group class	ACL_MASK	`mask::`*permissions*
Others	ACL_OTHER	`other::`*permissions*

The *permissions* are in the form `rwx`, with a hyphen (-) replacing any permissions that aren't granted. Here's an example of the ACL for a file:

```
user::rw-
user:violetta:r--
group::rw-
mask::rw-
other::---
```

The owner of the file has read and write permissions, as does the owning group. Others have no permissions at all. The user `violetta` has been granted read permission, so she's more privileged than “others”, but not quite as privileged as the owning user or group. If `violetta` hadn't been granted a special permission, the ACL would be the same as file permissions of `rw-rw----`.

If an ACL can be represented simply as file permissions, it's called a *minimal ACL*; if it can't, it's called an *extended ACL*. An extended ACL always has a mask entry, and can include any number of entries for named users and named groups. If a file or directory has an extended ACL, its permissions in the output of `ls -l` are followed by a plus sign (+).

The mask entry is the union of the permissions for the owning group, all named users, and all named groups. For example, let's consider a file whose owning group has no write permission:

```
# ls -l file.txt
-rw-r--r--   1 mabel    techies          50 Sep 27 21:22 file.txt
```

If we use the `getfacl` utility to get its ACL, we see:

```
# getfacl -q file.txt
user::rw-
group::r--
other::r--
```

The `-q` option suppresses some comments, listing the file name, owner, and group, that `getfacl` displays by default. Next, let's suppose that **mabel** uses `setfacl` to add an entry for **frank** that grants him read and write permission (in order to modify the ACL, you must be the owner of the file or directory, or have appropriate privileges):

```
# setfacl -m u:frank:rw- file.txt
# getfacl -q file.txt
user::rw-
user:frank:rw-
group::r--
mask::rw-
other::r--
# ls -l file.txt
-rw-rw-r--+  1 mabel    techies          50 Sep 27 21:22 file.txt
```

In addition to the entry for **frank**, the ACL now includes a mask entry that lists read and write permission. The output of `ls` also indicates read and write permission for the group.

Modifying the file permissions (e.g., using `chmod`) can affect the ACLs, and vice versa:

- The user file permissions and the permissions in the owning user ACL entry always match.
- The “other” file permissions and the permissions in the “other” ACL entry always match.
- If the ACL doesn't have a mask entry, the group file permissions and the permissions in the owning group ACL entry match.
- If the ACL has a mask entry, its permissions match the group file permissions. In this case, the owning group ACL entry's permissions aren't necessarily the same as the group file permissions.

Let's continue with the same sample file. Now, let's have **mabel** use `chmod` to remove write permission for the group:

```
# chmod g-w file.txt
# getfacl -q file.txt
```

```
user::rw-
user:frank:rw-          # effective: r--
group::r--
mask::r--
other::r--
# ls -l file.txt
-rw-r--r--+  1 mabel    techies          50 Sep 27 21:22 file.txt
```

The entry for **frank** still lists read and write permission, but a comment warns us that his effective permissions are read only, because we explicitly removed write permission from the mask.

The following pseudo-code shows the algorithm for checking the access to a file or directory:

```
if (the process's effective user ID matches the object owner's user ID)
{
   The matched entry is the owner ACL entry
}
else if (the process's effective user ID matches the user ID specified in any
         named user ACL entry)
{
   The matched entry is the matching named user entry
}
else if (the process's effective group ID or any of its supplementary group IDs
         matches the group ID of the object or matches the group ID
         specified in any named group entry)
{
   if (the requested access modes are granted by at least one entry
       matched by the process's effective group ID or any of its supplementary
       group IDs)
   {
       The matched entry is one of the granting entries (it doesn't matter which)
   }
   else
   {
       Access is denied
   }
}
else if (the requested access modes are granted by the “other” ACL entry)
{
   The matched entry is the “other” entry
}

if (the requested access modes are granted by the matched entry)
{
   if (the matched entry is the owning user or “other” entry)
   {
       Access is granted
   }
   else if (the requested access modes are also granted by the mask entry,
            or no mask entry exists in the ACL)
   {
       Access is granted
   }
   else
   {
        Access is denied
    }
}
else
{
    Access is denied
}
```

For more information about `getfacl` or `setfacl`, see the *Utilities Reference*.

There are also functions that you can use to work with ACLs in your programs; for information about them, see “Working with Access Control Lists” in the QNX Neutrino *Programmer's Guide*, and the *acl_*()* entries in the QNX Neutrino *C Library Reference*.

- The POSIX draft also describes *default ACLs* that specify the initial ACL for new objects created within a directory. Default ACLs aren't currently implemented.
- The `cp` utility doesn't copy any ACL that the source file has, but if the destination file already exists and has an ACL, its ACL is preserved.

Filename extensions

This table lists some common filename extensions used in a QNX Neutrino system.

Extension	Description	Related programs/utilities
.1	Troff-style text, e.g., from Unix "man" (manual) pages.	`man` and `troff` (third-party software)
.a	Library archive	`ar`
.awk	Awk script	`gawk`
.b	Bench calculator library or program	`bc`
.bat	MS-DOS batch file	For use on DOS systems; won't run under QNX Neutrino. See *Writing Shell Scripts* and `ksh` for information on writing shell scripts for QNX Neutrino.
.build	OS image buildfile	`mkifs`
.c	C program source code	`qcc`, `make`
.C, **.cc**, **.cpp**	C++ program source code	`q++`, `make`
.cfg	Configuration files, various formats	Various programs; formats differ
.conf	Configuration files, various formats	Various programs; formats differ
.css	Cascading style sheet	Used in the IDE for documentation
.def	C++ definition file	`q++`, `make`
.dll	MS-Windows dynamic link library	Not used directly in QNX Neutrino; necessary in support of some programs that run under MS-Windows, such as some of the QNX Neutrino tools. See **.so** (shared objects) for the QNX Neutrino equivalent.
.gz	Compressed file	`gzip`; *Backing Up and Recovering Data*
.h	C header file	`qcc`, `make`
.htm	HyperText Markup Language (HTML) file for Web viewing	Web browser
.ifs, **.img**	A QNX Neutrino Image filesystem, typically a bootable image	`mkifs`; see also OS Images in *Building Embedded Systems*

Extension	Description	Related programs/utilities
.jar	Java archive, consisting of multiple java files (class files etc.) compressed into a single file	Java applications e.g., the IDE
.kev	Kernel events, gathered by the instrumented kernel and used to profile an entire QNX Neutrino system	`procnto*-instr`, `tracelogger`, `traceprinter`, the IDE; see also the System Analysis Toolkit *User's Guide*
.mk	Makefile source, typically used within QNX Neutrino recursive makes	`make`
.o	Binary output file that results from compiling a C, C++, or Assembly source file	`qcc`, `make`
.S, **.s**	Assembly source code file	GNU assembler `as`
.so, **.so.***n*	Shared object	`qcc`, `make`
.tar	Tape archive	`tar`; *Backing Up and Recovering Data*
.tar.gz, **.tgz**	Compressed tape archive	`gzip`, `tar`; *Backing Up and Recovering Data*
.txt	ASCII text file	Many text-based editors, applications, and individual users
.use	Usage message source for programs that don't embed usage in the program source code (QNX Neutrino recursive make)	`make`
.wav	Audio wave file	
.xml	Extensible Markup Language file; multiple uses, including IDE documentation	
.zip	Compressed archive file	`gzip`

If you aren't sure about the format of a file, use the `file` utility:

```
file filename
```

Troubleshooting

Here are a few problems that you might have with files.

I'm trying to write a file, but I get a "permission denied" message.

You don't have write permission for the file. If you're the owner (or **root**) you can change the permissions; see "*File ownership and permissions*," above.

I'm trying to list a directory that I have write permission for, but I get a "permission denied" message.

You need to have read permission for a directory in order to list it, and execute permission in order to interact with the items below that directory. See "*File ownership and permissions*," above.

I'm having trouble with a file that has a space in its name.

The command interpreter, or shell, parses the command line and uses the space character to break the command into tokens. If your filename includes a space, you need to "quote" the space so that the shell knows you want a literal space. For more information, including other special characters that you need to watch for, see "*Quoting special characters*" in Using the Command Line.

Chapter 6 Using Editors

An editor is a utility designed to view and modify files.

Editors don't apply any persistent formatting to viewed text, although many use colors or styles to provide additional contextual information, such as type information in source code files. For example, if you're editing C code, some editors use different colors to indicate keywords, strings, numbers, and so on.

Supported editors

The QNX Neutrino RTOS includes and supports these editors:

`vi` (or `elvis`)

A powerful, but somewhat cryptic text-based editor that you'll find in most—if not all—Unix-style operating systems. It's actually the Visual Interface to an editor called `ex`.

On QNX Neutrino, `vi` is a symbolic link to `elvis`. To start `vi`, type:

```
vi filename
```

The `vi` editor has two modes:

Command mode

The keyboard is mapped to a set of command shortcuts used to navigate and edit text; `vi` commands consist of one or more letters, but `ex` commands start with a colon (`:`).

Insert mode

Lets you type normally.

To switch to command mode, press **Esc**; to switch to input mode, press one of:

- `I` or `i` to insert at the beginning of the current line or before the cursor
- `A` or `a` to append text at the end of the current line or after the cursor
- `O` or `o` to open a new line above or below the cursor

The two modes can make `vi` very confusing for a new user; by default, `vi` doesn't tell you which mode you're in. If you type this when you're in command mode:

```
:set showmode
```

the editor indicates the current mode, in the lower right corner of the display. If you always want this option set, you can add this command—without the colon—to the profile for `vi`, **$*HOME*/.exrc**.

Here are some of the `vi` commands that you'll use a lot:

To:	Press:
Leave `vi` without saving any changes	`:q!`
Save the current file	`:w`
Save the current file, and then exit	`:wq`, `:x`, or `ZZ`
Move the cursor to the left	`h` (see below)
Move the cursor to the right	`l` (see below)

To:	Press:
Move the cursor up one line	`k` (see below)
Move the cursor down one line	`j` (see below)
Move to the beginning of the next word	`w`
Move to the end of the current or next word (depending on the cursor position)	`e`
Move to the beginning of the current or previous word (depending on the cursor position)	`b`
Page back	**Ctrl–B**
Page forward	**Ctrl–F**
Yank (copy) the current line	`yy`
Yank from the cursor to the end of the current word	`yw`
Delete from the cursor to the end of the current word	`dw`
Delete the current line	`dd`
Paste text before the cursor	`P`
Paste text after the cursor	`p`

In some implementations of `vi`—including QNX Neutrino's—you can also use the arrow keys to move the cursor, whether you're in command or input mode.

You can combine the commands to make them even more useful; for example, type a number before `dd` to delete several lines at once. In addition, `vi` has 26 named buffers that let you easily cut or copy and paste different blocks of text.

You can find numerous resources, tutorials, and command summaries online. In QNX Neutrino, `vi` is actually a link to `elvis`; see the *Utilities Reference*.

Integrated Development Environment (IDE) editors

On Linux and Windows, the Integrated Development Environment (IDE) incorporates various specialized editors for creating C and C++ programs, buildfiles, and so on. You can use the Target File System Navigator in the IDE to edit files on your QNX Neutrino system. For more information, see the IDE *User's Guide*.

The Bazaar project on our Foundry27 website (*http://community.qnx.com*) may include other editors (as well as other third-party software that you might find useful). Note that we don't support these editors.

Specifying the default editor

Some system processes ask you to use an editor to provide some information. For example, if you check something into a version-control system, you're asked to explain the changes you made. Such processes use the *VISUAL* or *EDITOR* environment variable—or both—to determine which editor to use; the default is `vi`.

Historically, you used *EDITOR* to specify a line-oriented editor, and *VISUAL* to specify a fullscreen editor. Applications might use one or or both of these variables. Some applications that use both use *VISUAL* in preference to *EDITOR* when a fullscreen editor is required, or *EDITOR* in preference to *VISUAL* when a line-oriented editor is required.

Few modern applications invoke line-oriented editors, and few users set *EDITOR* to one, so you can't rely on applications to give preference one way or the other. For most uses, we recommend that you set *VISUAL* and *EDITOR* to the same value.

Once you've tried various editors, you can set these environment variables so that your favorite editor becomes the default. At the command-line prompt, type:

```
export VISUAL=path
export EDITOR=path
```

where *path* is the path to the executable for the editor.

To check the value of the *EDITOR* environment variable, type:

```
echo $EDITOR
```

You'll likely want to set these variables in your profile, **$*HOME*/.profile**, so that they're set whenever you log in. For more information, see "*$HOME/.profile*" in Configuring Your Environment.

Chapter 7
Configuring Your Environment

The Controlling How QNX Neutrino Starts chapter describes what happens when you boot your system, and what you can do to customize the system. This chapter describes how you can customize the environment that you get when you log in, and then describes some of the setup you might need to do.

What happens when you log in?

Before you start customizing your login environment, you should understand just what happens when you log in, because the nature of the customization determines where you should make it. You should consider these questions:

- Does this change apply to *all* users, or just to me?
- Do I need to do something only when I first log in, or whenever I start a shell?

When you log in, the system starts the login shell that's specified in your entry in the account database (see "*/etc/passwd*" in Managing User Accounts). The login shell is typically `sh`, which is usually just a link to the Korn shell, `ksh`.

When `ksh` starts as a login shell, it executes these profiles, if they exist and are executable:

- **/etc/profile**
- **$*HOME*/.profile**

Why have *two* profiles? Settings that apply to all users go into **/etc/profile**; your own customizations go into your own **.profile**. As you might expect, you need to be **root** to edit **/etc/profile**.

There's actually a third profile for the shell. The special thing about it is that it's executed whenever you start a shell; see "*`ksh`'s startup file*," below.

Customizing your home

Your home directory is where you can store all the files and directories that are relevant to you. It's a good place to store your own binaries and scripts.

Your entry in the password database specifies your home directory (see */etc/passwd* in Managing User Accounts), and the *HOME* environment variable stores this directory's name.

Your home directory is also where you store information that configures your environment when you log in. By default, applications pick this spot to install configuration files. Configuration files are generally preceded by a period (`.`) and run either when you log in (such as **.profile**) or when you start an application.

Configuring your shell

There are many files that configure your environment; this section describes some of the more useful ones.

- */etc/profile*
- *$HOME/.profile*
- *`ksh`'s startup file*

`/etc/profile`

The login shell executes **/etc/profile** if this file exists and is readable. This file does the shell setup that applies to all users, so you'll be interested in it if you're the system administrator; you need to log in as **root** in order to edit it.

The **/etc/profile** file:

- sets the *HOSTNAME* and *SYSNAME* environment variables if they aren't already set
- adds the appropriate directories to the *PATH* environment variable (the **root** user's *PATH* includes directories such as **/sbin** that contain system executables)
- sets up the file-permission mask (`umask`); see "*File ownership and permissions*" in Working with Files
- displays the date you logged in, the "message of the day" (found in **/etc/motd**), and the date you last logged in
- sets the *TMPDIR* environment variable to **/tmp** if it isn't already set.
- runs any scripts in the **/etc/profile.d** directory as "dot" files (i.e., instead of executing them as separate shells, the current shell loads their commands into itself). For more information about dot files, see "**.** (dot) builtin command" in the documentation for `ksh` in the *Utilities Reference*.

If you have a script that you want to run whenever anyone on the system runs a login shell, put it in the **/etc/profile.d** directory. You must have **root**-level privileges to add a file to this directory.

For example, if you need to set global environment variables or run certain tasks when anyone logs in, then this is the place to put a script to handle it. If you're using `sh` as your login shell, make sure that the script has a **.sh** extension.

$*HOME*/`.profile`

The system runs **$HOME/.profile** whenever you log in, after it runs **/etc/profile**. If you change your **.profile**, the changes don't go into effect until you next log in.

You should use your **.profile** to do the customizations that you need to do only once, or that you want all shells to inherit. For example, you could:

- set environment variables; see "*Environment variables*"
- run any commands that *you* need
- set your file-permission mask; see "*File ownership and permissions*" in Working with Files

If you want to create an alias, you should do it in your shell's profile (see "*ksh's startup file*"), not in **.profile**, because the shell doesn't export aliases. If you do set an alias in **.profile**, the alias is set only in shells that you start as login shells, using the `-l` option.

For an example of **.profile**, see the *Examples* appendix.

`ksh`'s startup file

As described above, the login shell runs certain profiles. In addition, you can have a profile that `ksh` runs whenever you start a shell—whether or not it's a login shell.

This profile doesn't have a specific name; when you start `ksh`, it checks the *ENV* environment variable. If this variable exists, `ksh` gets the name of the profile from it. To set up *ENV*, add a line like this to your *$HOME/.profile* file:

```
export ENV=$HOME/.kshrc
```

People frequently call the profile **.kshrc**, but you can give it whatever name you want. This file doesn't need to be executable.

Use `ksh`'s profile to set up your favorite aliases, and so on. For example, if you want `ls` to always display characters that tell you if a file is executable, a directory, or a link, add this line to the shell's profile:

```
alias ls="ls -F"
```

Any changes that you make to the profile apply to new shells, but not to existing instances.

For an example of **.kshrc**, see the *Examples* appendix.

Environment variables

Many applications use environment variables to control their behavior.

For example, `less` gets the width of the terminal or window from the *COLUMNS* environment variable; many utilities write any temporary files in the directory specified by *TMPDIR*. For more information, see the Commonly Used Environment Variables appendix of the *Utilities Reference*.

When you start a process, it inherits a copy of its parent's environment. This means that you can set an environment variable in your **.profile**, and all your shells and processes inherit it—provided that no one in the chain undefines it.

For example, if you have your own **bin** directory, you can add it to your *PATH* by adding a line like this to your **.profile**:

```
export PATH=$PATH:/home/username/bin
```

If you're the system administrator, and you want this change to apply to everyone, export the environment variables from **/etc/profile** or from a script in **/etc/profile.d**. For more information, see the discussion of */etc/profile* earlier in this chapter.

Setting *PATH* and *LD_LIBRARY_PATH*

The `login` utility doesn't preserve environment variables, except for a few special ones, such as `PATH` and `TERM`.

The *PATH* environment variable specifies the search paths for commands, while *LD_LIBRARY_PATH* specifies the search paths for shared libraries for the linker.

The initial default values of *PATH* and *LD_LIBRARY_PATH* are specified in the buildfile before `procnto` is started. Two configuration strings (see "*Configuration strings*," below), _CS_PATH and _CS_LIBPATH, take the default values of *PATH* and *LD_LIBRARY_PATH*. The `login` utility uses _CS_PATH to set the value of *PATH* and passes this environment variable and both configuration strings to its child processes.

If you type `set` or `env` in a shell that was started from `login`, you'll see the *PATH* variable, but not *LD_LIBRARY_PATH*; _CS_LIBPATH works in the same manner as *LD_LIBRARY_PATH*.

You can use the **/etc/default/login** file to indicate which environment variables you want `login` to preserve. You can edit this file to add new variables, such as *LD_LIBRARY_PATH*, but you can't change existing variables such as *PATH* and *TERM*.

If you use `ksh` as your login shell, you can edit **/etc/profile** and **$HOME/.profile** to override existing variables and add new ones. Any environment variables set in **/etc/profile** override previous settings in **/etc/default/login**; and **$HOME/.profile** overrides both **/etc/default/login** and **/etc/profile**.

For more information on configuration strings, see "*Configuration strings*," below.

Configuration strings

In addition to environment variables, QNX Neutrino uses *configuration strings*. These are system variables that are like environment variables, but are more dynamic.

When you set an environment variable, the new value affects only the current instance of the shell and any of its children that you create after setting the variable; when you set a configuration string, its new value is immediately available to the entire system.

QNX Neutrino also supports *configurable limits*, which are variables that store information about the system. For more information, see the *Understanding System Limits* chapter.

You can use the POSIX `getconf` utility to get the value of a configurable limit or a configuration string. QNX Neutrino also defines a non-POSIX `setconf` utility that you can use to set configuration strings if you're logged in as **root**. In a program, call *confstr()* to get the value of a configuration string.

The names of configuration strings start with `_CS_` and are in uppercase, although `getconf` and `setconf` let you use any case, omit the leading underscore, or the entire prefix—provided that the rest of the name is unambiguous.

The full list of supported configuration strings is given in the reference for the *confstr()* QNX Neutrino library function. The supported strings include at least the following:

_CS_ARCHITECTURE

The name of the instruction-set architecture.

_CS_CONFIG_PATH

A colon-separated list of directories to search for configuration files.

_CS_DOMAIN

The domain of this node in the network.

_CS_HOSTNAME

The name of this node in the network.

A hostname can consist only of letters, numbers, and hyphens, and must not start or end with a hyphen. For more information, see *RFC 952*.

If you change this configuration string, be sure you also change the *HOSTNAME* environment variable. The `hostname` utility always gives the value of the _CS_HOSTNAME configuration string.

_CS_HW_PROVIDER

The name of the hardware's manufacturer.

_CS_HW_SERIAL

The serial number associated with the hardware.

_CS_LIBPATH

The default path for locating shared objects. For more information, see "*Setting PATH and LD_LIBRARY_PATH* ."

_CS_LOCALE

The locale string.

_CS_MACHINE

The type of hardware the OS is running on.

_CS_PATH

The default path for finding system utilities. For more information, see "*Setting PATH and LD_LIBRARY_PATH* ."

_CS_RELEASE

The current release level of the OS.

_CS_RESOLVE

An in-memory version of the **/etc/resolv.conf** file, excluding the domain name.

_CS_SRPC_DOMAIN

The secure RPC (Remote Procedure Call) domain.

_CS_SYSNAME

The name of the OS.

_CS_TIMEZONE

An alternate source to the *TZ* for time-zone information. For more information, see "*Setting the time zone*," below.

_CS_VERSION

The version of the OS.

Setting the time zone

On the command line, you can set the time zone by setting the *TZ* environment variable or the _CS_TIMEZONE configuration string.

If *TZ* isn't set, the system uses the value of the _CS_TIMEZONE configuration string instead. The POSIX standards include the *TZ* environment variable; _CS_TIMEZONE is a QNX Neutrino implementation. The description below applies to both.

Various time functions use the time-zone information to compute times relative to Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time (GMT). You usually set the time on your computer to UTC. Use the `date` command if the time isn't automatically maintained by the computer hardware.

You can set the *TZ* environment variable by using the `env` utility or the `export` shell command. You can use `setconf` to set _CS_TIMEZONE. For example:

```
env TZ=PST8PDT
export TZ=PST8PDT
setconf _CS_TIMEZONE PST8PDT
```

The *TZ* environment variable or _CS_TIMEZONE string can be in POSIX or `zoneinfo` format:

- POSIX format is as follows (spaces are for clarity only):

  std offset dst offset, rule

  The expanded format is as follows:

  stdoffset[dst[offset][,start[/time],end[/time]]]

  The components are:

  std* and *dst

  Three or more letters that you specify to designate the standard or daylight saving time zone. Only *std* is required. If you omit *dst*, then daylight saving time doesn't apply in this locale. Upper- and lowercase letters are allowed. Any characters except for a leading colon (:), digits, comma (,), minus (-), plus (+), and ASCII NUL (\0) are allowed.

  offset

  The value you must add to the local time to arrive at Coordinated Universal Time (UTC). The *offset* has the form:

  hh[**:***mm*[**:***ss*]]

  Minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required; it may be a single digit.

  The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time.

  You can use one or more digits; the value is always interpreted as a decimal number. The hour may be between 0 and 24; the minutes (and seconds), if present, between 0 and 59. If preceded by a "−", the time zone is east of the prime meridian; otherwise it's west (which may be indicated by an optional preceding "+").

rule

Indicates when to change to and back from summer time. The *rule* has the form:

date / *time* **,** *date* / *time*

where the first *date* describes when the change from standard to summer time occurs, and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* may be one of the following:

J*n*

The Julian day *n* (1 <= n <= 365). Leap days aren't counted. That is, in all years—including leap years—February 28 is day 59 and March 1 is day 60. It's impossible to refer explicitly to the occasional February 29.

n

The zero-based Julian day (0 <= n <= 365). Leap years are counted; it's possible to refer to February 29.

M*m.n.d*

The *d*th day (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12, where week 5 means "the last *d* day in month *m*", which may occur in the fourth or fifth week). Week 1 is the first week in which the *d*th day occurs. Day zero is Sunday.

The *time* has the same format as *offset*, except that no leading sign ("+" or "-") is allowed. The default, if *time* is omitted, is 02:00:00.

- The `zoneinfo` format is as follows:

  :*area*/*location*

  For example, `:Asia/Tokyo` corresponds to Japanese Standard Time.

Time zones specified in this format require a database that you can obtain from the Internet Assigned Numbers Authority at *https://www.iana.org/time-zones*. Download and `untar` the archive on your development host, build it following the instructions in the **README** file, and include the compiled database on your target system. You must install the database in **/usr/share/zoneinfo**.

Examples

Let's look at some examples of time zones.

Eastern time

The default time zone is Eastern time; the short specification is:

```
EST5EDT
```

The full specification is:

```
EST5EDT4,M3.2.0/02:00:00,M11.1.0/02:00:00
```

Both are interpreted as follows:

- Eastern Standard Time is 5 hours earlier than Coordinated Universal Time (UTC). Standard time and daylight saving time both apply to this locale.
- By default, Eastern Daylight Time (EDT) is one hour ahead of standard time (i.e., EDT4).
- Daylight saving time starts on the second (2) Sunday (0) of March (3) at 2:00 A.M. and ends on the first (1) Sunday (0) of November (11) at 2:00 A.M.

Pacific time

The short specification for Pacific time is:

```
PST8PDT
```

The full specification is:

```
PST08PDT07,M3.2.0/2,M11.1.0/2
```

Both are interpreted as follows:

- Pacific Standard Time is 8 hours earlier than Coordinated Universal Time (UTC).
- Standard time and daylight saving time both apply to this locale.
- By default, Pacific Daylight Time is one hour ahead of standard time (that is, PDT7).
- Daylight saving time starts on the second (2) Sunday (0) of March (3) at 2:00 A.M. and ends on the first (1) Sunday (0) of November (11) at 2:00 A.M.

Newfoundland time

The short specification for Newfoundland time is:

```
NST3:30NDT2:30
```

The full specification is:

```
NST03:30NDT02:30,M3.2.0/00:01,M11.1.0/00:01
```

Both are interpreted as follows:

- Newfoundland Standard Time is 3.5 hours earlier than Coordinated Universal Time (UTC).

- Standard time and daylight saving time both apply to this locale.
- Newfoundland Daylight Time is 2.5 hours earlier than Coordinated Universal Time (UTC).
- Daylight saving time starts on the second (2) Sunday (0) of March (3) at 12:01:00 A.M. and ends on the first (1) Sunday (0) of November (11) at 12:01:00 A.M.

Central European time

The specification for Central European time is:

```
Central Europe Time-2:00
```

- Central European Time is 2 hours later than Coordinated Universal Time (UTC).
- Daylight saving time doesn't apply in this locale.

Japanese time

The specification for Japanese time is:

```
JST-9
```

- Japanese Standard Time is 9 hours earlier than Coordinated Universal Time (UTC).
- Daylight saving time doesn't apply in this locale.

Programming with time zones

Inside a program, you can set the *TZ* environment variable by calling *setenv()* or *putenv()*.

For example:

```
setenv( "TZ", "PST08PDT07,M3.2.0/2,M11.1.0/2", 1 );
putenv( "TZ=PST08PDT07,M3.2.0/2,M11.1.0/2" );
```

To obtain the value of the variable, use the *getenv()* function:

```
char *tzvalue;
…
tzvalue = getenv( "TZ" );
```

You can get the value of _CS_TIMEZONE by calling *confstr()*, like this:

```
confstr( _CS_TIMEZONE, buff, BUFF_SIZE );
```

or set it like this:

```
confstr( _CS_SET | _CS_TIMEZONE, "JST-9", 0 );
```

The *tzset()* function gets the current value of *TZ*—or _CS_TIMEZONE if *TZ* isn't set—and sets the following global variables:

daylight

Indicates if daylight saving time is supported in the locale.

timezone

The number of seconds of time difference between the local time zone and Coordinated Universal Time (UTC).

tzname

A vector of two pointers to character strings containing the standard and daylight time zone names.

Whenever you call *ctime()*, *ctime_r()*, *localtime()*, or *mktime()*, the library sets *tzname*, as if you had called *tzset()*. The same is true if you use the `%Z` directive when you call *strftime()*.

For more information about these functions and variables, see the QNX Neutrino *C Library Reference*.

Terminal types

You need to set the *TERM* environment variable to indicate to your console what type of terminal you're using.

The **/usr/lib/terminfo** directory contains directories that contain terminal database information. You can use the utilities `tic` and `infocmp` to change the mappings in the database.

For example, you could run `infocmp` on **/usr/lib/terminfo/q/qansi-m**, and this would generate the source for this database. You could then modify the source and then run the `tic` utility on that source to compile the source back in to a reconcilable database. The **/etc/termcap** file is provided for compatibility with programs that use the older single-file database model as opposed to the newer library database model.

For more information, see:

Strang, John, Linda Mui, and Tim O'Reilly. 1988. *termcap & terminfo*. Sebastopol, CA: O'Reilly and Associates. ISBN 0937175226.

Troubleshooting

Here are some common problems you might encounter while customizing your environment:

A script I put in* /etc/profile.d *doesn't run.

Check the following:

- Make sure that the script's name has **.ksh** or **.sh** as its extension.
- Make sure the executable bit is set on the script.
- Make sure that the script begins with the line:

```
#! /bin/sh
```

How do I set the time so it's right in QNX Neutrino* and *Microsoft Windows?

If you have Windows in one partition and QNX Neutrino in another on your machine, you might notice that setting the clock on one OS changes it on the other.

Under QNX Neutrino, you usually set the hardware clock to use UTC (Coordinated Universal Time) and then set the time zone. Under Windows, you set the hardware clock to use local time.

To set the time so that it's correct in both operating systems, set the hardware clock to use local time under QNX Neutrino.

How can I properly check if* .kshrc *is being run as a script rather than as a terminal session?

If the `i` option is set, then **.kshrc** is running in interactive mode. Here's some code that checks to see if this option is set:

```
case $- in
*i*)

     set -o emacs

     export EDITOR=vi
     export VISUAL=vi
     export PS1='`hostname -s`:`/bin/pwd` >'

     bind ^[[z=list
     bind ^I=complete

     ...
esac
```

The `$-` parameter is a concatenation of all the single-letter options that are set for the script. For more information, see “Parameters” in the entry for `ksh` in the *Utilities Reference*.

Chapter 8 Writing Shell Scripts

Shell scripting, at its most basic, is taking a series of commands you might type at a command line and putting them into a file, so you can reproduce them again at a later date, or run them repeatedly without having to type them over again.

You can use scripts to automate repeated tasks, handle complex tasks that might be difficult to do correctly without repeated tries, redoing some of the coding, or both.

Available shells

The shell that you'll likely use for scripting under QNX Neutrino is `ksh`, a public-domain implementation of the Korn shell. The `sh` command is usually a symbolic link to `ksh`.

For more information about this shell, see:

- the *Using the Command Line* chapter in this guide
- the entry for `ksh` in the *Utilities Reference*
- Rosenblatt, Bill, and Arnold Robbins. 2002. *Learning the Korn Shell*, 2nd Edition. Sebastopol, CA: O'Reilly & Associates. ISBN 0-596-00195-9

QNX Neutrino also supplies or uses some other scripting environments:

- An OS buildfile has a script file section tagged by `+script`. The `mkifs` parses this script, but it's executed by `procnto` at boot time. It provides a very simple scripting environment, with the ability to run a series of commands, and a small amount of synchronization.
- The embedded shell, `esh`, provides a scripting environment for running simple scripts in an embedded environment where the overhead of the full `ksh` might be too much. It supports the execution of utilities, simple redirection, filename expansion, aliases, and environment manipulation.
- The fat embedded shell, `fesh`, provides the same limited environment as `esh`, but supplies additional builtin commands for commonly used utilities to reduce the overhead of including them in an embedded system. The `fesh` shell includes builtins for `cp`, `df`, `ls`, `mkdir`, `rm`, and `rmdir`, although in most cases, the builtin provides only the core functionality of the utility and isn't a complete replacement for it.
- The micro-embedded shell, `uesh`, provides a subset of `esh`'s functionality, and is suitable for situations with very limited memory.
- `python` is a powerful object-oriented language that you can use for processing files, manipulating strings, parsing HTML, and much more.
- `sed` is a stream editor, which makes it most useful for performing repeated changes to a file, or set of files. It's often used for scripts, or as a utility within other scripts.
- `gawk` (GNU `awk`) is a programming language for pattern matching and working with the contents of files. You can also use it for scripting or call it from within scripts.
- The Bazaar project on our Foundry 27 website (*http://community.qnx.com*) includes `perl`, which, like `gawk`, is useful for working with files and patterns. The name `perl` stands for Practical Extraction and Report Language.

In general, a shell script is most useful and powerful when working with the execution of programs or modifying files in the context of the filesystem, whereas `sed`, `gawk`, and `perl` are primarily for working with the contents of files. For more information, see:

- the entries for `gawk` and `sed` in the *Utilities Reference*
- Robbins, Arnold, and Dale Dougherty. 1997. *sed & awk*, 2nd Edition. Sebastopol, CA: O'Reilly & Associates. ISBN 1-56592-225-5
- Schwartz, Randal L., and Tom Phoenix. 2001. *Learning Perl*. Sebastopol, CA: O'Reilly & Associates. ISBN 0-59600-132-0

Running a shell script

You can execute a shell script in these ways:

- Invoke another shell with the name of your shell script as an argument:

  ```
  sh myscript
  ```

- Load your script as a “dot file” into the current shell:

  ```
  . myscript
  ```

- Use `chmod` to make the shell script executable, and then invoke it, like this:

  ```
  chmod 744 myscript
  ./myscript
  ```

  In this instance, your shell automatically invokes a new shell to execute the shell script.

The first line

The first line of a script can identify the interpreter to use.

The first line of many—if not most—shell scripts is in this form:

```
#! interpreter [arg]
```

For example, a Korn shell script likely starts with:

```
#! /bin/sh
```

The line starts with a `#`, which indicates a comment, so the line is ignored by the shell processing this script. The initial two characters, `#!`, aren't important to the shell, but the loader code in `procnto` recognizes them as an instruction to load the specified interpreter and pass it:

1. the path to the interpreter
2. the optional argument specified on the first line of the script
3. the path to the script
4. any arguments you pass to the script

For example, if your script is called `my_script`, and you invoke it as:

```
./my_script my_arg1 my_arg2 ...
```

then `procnto` loads:

```
interpreter [arg] ./my_script my_arg1 my_arg2 ...
```

- The interpreter can't be another `#!` script.
- The process manager ignores any setuid and getuid permissions on the script; the child still has the same user and group IDs as its parent. (For more information, see "*Setuid and setgid*" in the Working with Files chapter of this guide.)

Some interpreters adjust the list of arguments:

- `ksh` removes itself from the arguments
- `gawk` changes its own path to be simply `gawk`
- `perl` removes itself and the name of the script from the arguments, and puts the name of the script into the `$0` variable

For example, let's look at some simple scripts that echo their own arguments.

Arguments to a ksh script

Suppose we have a script called ksh_script that looks like this:

```
#! /bin/sh
echo $0
for arg in "$@" ; do
  echo $arg
done
```

If you invoke it as `./ksh_script one two three`, the loader invokes it as `/bin/sh ./ksh_script one two three`, and then `ksh` removes itself from the argument list. The output looks like this:

```
./ksh_script
one
two
three
```

Arguments to a `gawk` script

Next, let's consider the `gawk` version, **gawk_script**, which looks like this:

```
#!/usr/bin/gawk -f
BEGIN {
        for (i = 0; i < ARGC; i++)
                print ARGV[i]
}
```

The `-f` argument is important; it tells `gawk` to read its script from the given file. Without `-f`, this script wouldn't work as expected.

If you run this script as `./gawk_script one two three`, the loader invokes it as `/usr/bin/gawk -f ./gawk_script one two three`, and then `gawk` changes its full path to `gawk`. The output looks like this:

```
gawk
one
two
three
```

Arguments to a `perl` script

The perl version of the script, **perl_script**, looks like this:

```
#! /usr/bin/perl
for ($i = 0; $i <= $#ARGV; $i++) {
    print "$ARGV[$i]\n";
}
```

If you invoke it as `./perl_script one two three`, the loader invokes it as `/usr/bin/perl ./perl_script one two three`, and then `perl` removes itself and the name of the script from the argument list. The output looks like this:

```
one
two
three
```

Example of a Korn shell script

Let's look at a script that searches C source and header files in the current directory tree for a string passed on the command line:

```
#!/bin/sh
#
# tfind:
# script to look for strings in various files and dump to less

case $# in
1)
    find . -name '*.[ch]' | xargs grep $1 | less
    exit 0   # good status
esac

echo "Use tfind stuff_to_find                              "
echo "      where : stuff_to_find = search string          "
echo "                                                     "
echo "e.g., tfind console_state looks through all files in  "
echo "     the current directory and below and displays all "
echo "     instances of console_state."
exit 1    # bad status
```

As described above, the first line identifies the program, **/bin/sh**, to run to interpret the script. The next few lines are comments that describe what the script does. Then we see:

```
case $# in
1)
  ...
esac
```

The `case ... in` is a shell builtin command, one of the branching structures provided by the Korn shell, and is equivalent to the C `switch` statement.

The `$#` is a shell variable. When you refer to a variable in a shell, put a `$` before its name to tell the shell that it's a variable rather than a literal string. The shell variable, `$#`, is a special variable that represents the number of command-line arguments to the script.

The `1)` is a possible value for the case, the equivalent of the C `case` statement. This code checks to see if you've passed exactly one parameter to the shell.

The `esac` line completes and ends the `case` statement. Both the `if` and `case` commands use the command's name reversed to represent the end of the branching structure.

Inside the case we find:

```
find . -name '*.[ch]' | xargs grep $1 | less
```

This line does the bulk of the work, and breaks down into these pieces:

- `find . -name '*.[ch]'`
- `xargs grep $1`
- `less`

which are joined by the | or pipe character. A pipe is one of the most powerful things in the shell; it takes the output of the program on the left, and makes it the input of the program to its right. The pipe lets you build complex operations from simpler building blocks. For more information, see "*Redirecting input and output*" in Using the Command Line.

The first piece, `find . -name '*.[ch]'`, uses another powerful and commonly used command. Most filesystems are recursive through a hierarchy of directories, and `find` is a utility that descends through the hierarchy of directories recursively. In this case, it searches for files that end in either **.c** or **.h**—that is, C source or header files—and prints out their names.

The filename wildcards are wrapped in single quotes (`'`) because they're special characters to the shell. Without the quotes, the shell would expand the wildcards in the current directory, but we want `find` to evaluate them, so we prevent the shell from evaluating them by quoting them. For more information, see "*Quoting special characters*" in Using the Command Line.

The next piece, `xargs grep $1`, does a couple of things:

- `grep` is a file-contents search utility. It searches the files given on its command line for the first argument. The `$1` is another special variable in the shell that represents the first argument we passed to the shell script (i.e., the string we're looking for).
- `xargs` is a utility that takes its input and turns it into command-line parameters for some other command that you give it. Here, it takes the list of files from `find` and makes them command-line arguments to `grep`. In this case, we're using `xargs` primarily for efficiency; we could do something similar with just `find`:

  ```
  find . -name '*.[ch]' -exec grep $i {} | less
  ```

  which loads and runs the `grep` program for every file found. The command that we actually used:

  ```
  find . -name '*.[ch]' | xargs grep $1 | less
  ```

  runs `grep` only when `xargs` has accumulated enough files to fill a command line, generally resulting in far fewer invocations of `grep` and a more efficient script.

The final piece, `less`, is an output pager. The entire command may generate a lot of output that might scroll off the terminal, so `less` presents this to you a page at a time, with the ability to move backwards and forwards through the data.

The `case` statement also includes the following after the `find` command:

```
exit 0   # good status
```

This returns a value of 0 from this script. In shell programming, zero means true or success, and anything nonzero means false or failure. (This is the opposite of the meanings in the C language.)

The final block:

```
echo "Use tfind stuff_to_find       "
echo "      where : stuff_to_find = search string           "
echo "      "
echo "e.g., tfind console_state looks through all files in  "
echo "     the current directory and below and displays all "
echo "     instances of console_state."
exit 1    # bad status
```

is just a bit of help; if you pass incorrect arguments to the script, it prints a description of how to use it, and then returns a failure code.

Efficiency

In general, a script isn't as efficient as a custom-written C or C++ program, because it:

- is interpreted, not compiled
- does most of its work by running other programs

However, developing a script can take less time than writing a program, especially if you use pipes and existing utilities as building blocks in your script.

Caveat scriptor

If you need to write shell scripts, there are a few things to bear in mind.

- In order to run a script as if it were a utility, you must make it executable by using the `chmod` command. For example, if you want anyone to be able to run your script, type:

  ```
  chmod a+x script_name
  ```

  Your script doesn't have to be executable if you plan to invoke it by passing it as a shell argument:

  ```
  ksh script_name
  ```

  or if you use it as a “dot file,” like this:

  ```
  . script_name
  ```

- Just as for any executable, if your script isn't in one of the directories in your *PATH*, you have to specify the path to the script in order to run it. For example:

  ```
  ~/bin/my_script
  ```

- When you run a script, it inherits its environment from the parent process. If your script executes a command that might not be in the *PATH*, you should either specify the path to the command or add the path to the script's *PATH* variable.
- A script can't change its parent shell's environment or current directory, unless you run it as a dot file.
- A script won't run if it contains DOS end-of-line characters. If you edit a QNX Neutrino script on a Windows machine, use the `textto` utility with the `-l` option to convert the file to the format used by the Power-Safe filesystem.

Chapter 9 Working with Filesystems

The QNX Neutrino RTOS provides a variety of filesystems, so that you can easily access DOS, Linux, as well as native Power-Safe disks.

The Filesystems chapter of the *System Architecture* guide describes their classes and features.

Under QNX Neutrino:

- You can dynamically start and stop filesystems.
- Multiple filesystems may run concurrently.
- Applications are presented with a single unified pathname space and interface, regardless of the configuration and number of underlying filesystems.

A desktop QNX Neutrino system starts the appropriate block filesystems on booting; you start other filesystems as standalone managers. The default block filesystem is the *Power-Safe filesystem*.

Setting up, starting, and stopping a block filesystem

When you launch a block device driver (`devb-*`), it detects the partitions on the block I/O devices. You can then issue `mount` commands to start the appropriate filesystems for various partitions.

You aren't likely ever to need to stop or restart a block filesystem; if you change any of the filesystem's options, you can use the `-e` or `-u` option to the `mount` command to update the filesystem.

If you need to change any of the options associated with the block I/O device, you can `slay` the appropriate `devb-*` driver (being careful not to pull the carpet from under your feet) and restart it, but you'll need to explicitly mount any of the filesystems on it.

You can determine the maximum filename length that a filesystem supports by using the `getconf` utility:

```
getconf _PC_NAME_MAX root_dir
```

where *root_dir* is the root directory of the filesystem.

To determine how much free space you have on a filesystem, use the `df` command. For more information, see the *Utilities Reference*.

Some filesystems have the concept of being marked as “dirty.” This can be used to skip an intensive filesystem-check the next time it starts up. The Ext2 filesystem has a flag bit; the DOS filesystem has some magic bits in the FAT. By default, when you mount a filesystem as read-write, that flag is set; when you cleanly unmount the filesystem, the flag is cleared. In between, the filesystem is dirty and may need to be checked (if it never gets cleanly unmounted). The Power-Safe filesystem has no such flag; it just rolls back to the last clean snapshot. You can use the `blk marking=none` option to turn off this marking; see the entry for **io-blk.so** in the *Utilities Reference*.

Mounting and unmounting filesystems

The following utilities work with filesystems:

`mount`

Mount a block-special device or remote filesystem.

`umount`

Unmount a device or filesystem.

For example, if `fs-cifs` is already running, you can mount filesystems on it like this:

```
mount -t cifs -o guest,none //SMB_SERVER:10.0.0.1:/QNX_BIN /bin
```

By default, filesystems are mounted as read-write if the physical media permit it. You can use the `-r` option for `mount` to mount the filesystem as read-only. The **io-blk.so** library also supports an `ro` option for mounting block I/O filesystems as read-only.

You can also use the `-u` option for the `mount` utility to temporarily change the way the filesystem is mounted. For example, if a filesystem is usually mounted as read-only, and you need to remount it as read-write, you can update the mounting by specifying `-uw`. For example:

```
mount -uw /
```

To return to read-only mode, use the `-ur` options:

```
mount -ur /
```

You should use `umount` to unmount a read-write filesystem before removing or ejecting removable media.

See the *Utilities Reference* for details on usage and syntax.

Image filesystem

By an *image*, we refer to an OS image here, which is a file that contains the OS, your executables, and any data files that might be related to your programs, for use in an embedded system.

You can think of the image as a small “filesystem”; it has a directory structure and some files in it.

The image contains a small directory structure that tells `procnto` the names and positions of the files contained within it; the image also contains the files themselves. When the embedded system is running, the image can be accessed just like any other read-only filesystem:

```
# cd /proc/boot
# ls
.script     cat         data1       data2       devc-ser8250
esh         ls          procnto
# cat data1
This is a data file, called data1, contained in the image.
Note that this is a convenient way of associating data
files with your programs.
```

The above example actually demonstrates two aspects of having the OS image function as a filesystem. When we issue the `ls` command, the OS loads `ls` from the image filesystem (pathname **/proc/boot/ls**). Then, when we issue the `cat` command, the OS loads `cat` from the image filesystem as well, and opens the file **data1**.

You can create an OS image by using `mkifs` (MaKe Image FileSystem). For more information, see *Building Embedded Systems*, and `mkifs` in the *Utilities Reference*.

`/dev/shmem` RAM "filesystem"

QNX Neutrino provides a simple RAM-based "filesystem" that allows read/write files to be placed under **/dev/shmem**.

Note that **/dev/shmem** isn't actually a filesystem. It's a window onto the shared memory names that happens to have *some* filesystem-like characteristics.

Shared memory objects in the **/dev/shmem** directory are advertised as "name-special" files (S_IFNAM), which fools many utilities—such as `more`—that expect regular files (S_IFREG). For this reason, many utilities might not work for such files. Files placed there using normal command-line utilities look and act like normal files.

If you want to use `gzip` to compress or expand files in **/dev/shmem**, you need to specify the `-f` option.

This filesystem is mainly used by the shared memory system of `procnto`. In special situations (e.g., when no filesystem is available), you can use the RAM filesystem to store file data. There's nothing to stop a file from consuming all free RAM; if this happens, other processes might have problems.

You'll use the RAM filesystem mostly in tiny embedded systems where you need a small, fast, *temporary-storage* filesystem, but you don't need persistent storage across reboots.

The filesystem comes for free with `procnto` and doesn't require any setup or device driver. You can simply create files under **/dev/shmem** and grow them to any size (depending on RAM resources).

Although the RAM filesystem itself doesn't support hard or soft links or directories, you can create a link to it by using process-manager links. For example, you could create a link to a RAM-based **/tmp** directory:

```
ln -sP /dev/shmem /tmp
```

This tells `procnto` to create a process-manager link to **/dev/shmem** known as **/tmp**. Most application programs can then open files under **/tmp** as if it were a normal filesystem.

In order to minimize the size of the RAM filesystem code inside the process manager, this filesystem specifically doesn't include "big filesystem" features such as:

- file locking
- directories
- **.** and **..** entries for the current and parent directories
- hard or soft links
- protection from overwriting running executables. A real filesystem gives an error of EBUSY if you try this; in **/dev/shmem**, the running executable will likely crash. This is because being able to write to a shared memory object while somebody else has it open is the whole point of shared memory.

Power-Safe filesystem

The Power-Safe filesystem, supported by the **fs-qnx6.so** shared object, is a reliable disk filesystem that can withstand power failures without losing or corrupting data.

Its features include the following:

- 510-byte (UTF-8) filenames

  You can't use the characters `0x00-0x1F`, `0x7F`, and `0xFF` in filenames. In addition, `/` (`0x2F`) is the pathname separator, and can't be in a filename component. You can use spaces, but you have to "quote" them on the command line; you also have to quote any wildcard characters that the shell supports. For more information, see "*Quoting special characters*" in Using the Command Line.

- copy-on-write (COW) updates that prevent the filesystem from becoming corrupted by a power failure while writing
- a snapshot that captures a consistent view of the filesystem

For information about the structure of this filesystem, see "Power-Safe filesystem" in the Filesystems chapter of the *System Architecture* guide.

CAUTION: If the drive doesn't support synchronizing, **fs-qnx6.so** can't guarantee that the filesystem is power-safe. Before using this filesystem on devices—such as USB/Flash devices—other than traditional rotating hard disk drive media, check to make sure that your device meets the filesystem's requirements. For more information, see "Required properties of the device" in the entry for **fs-qnx6.so** in the *Utilities Reference*.

To create a Power-Safe filesystem, use the `mkqnx6fs` utility. For example:

```
mkqnx6fs /dev/hd0t76
```

You can use the `mkqnx6fs` options to specify the logical blocksize, endian layout, number of logical blocks, maximum number of inodes (and hence, files), and so on.

Once you've formatted the filesystem, simply `mount` it. For example:

```
mount -t qnx6 /dev/hd0t76 /mnt/psfs
```

For more information about the options for the Power-Safe filesystem, see **fs-qnx6.so** in the *Utilities Reference*.

To check the filesystem for consistency (which you aren't likely to need to do), use `chkqnx6fs`.

Links and inodes

File data is stored distinctly from its name and can be referenced by more than one name. Each filename, called a *link*, points to the actual data of the file itself.

(There are actually two kinds of links: *hard links*, which we refer to simply as "links," and *symbolic links*, which are described in the next section.)

In order to support links for each file, the filename is separated from the other information that describes a file. The non-filename information is kept in a storage table called an *inode* (for "information node").

If a file has only one link (i.e., one filename), the inode information (i.e., the non-filename information) is stored in the directory entry for the file. If the file has more than one link, the inode is stored as a record in a special file named **/.inodes**—the file's directory entry points to the inode record.

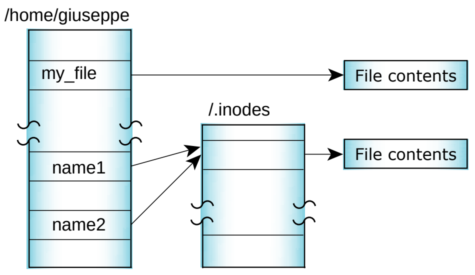


Figure 2: One file referenced by two links.

Note that you can create a link to a file only if the file and the link are in the same filesystem.

There are some situations in which a file can have an entry in the **/.inodes** file:

- If a file at one time had more than one link, and all links but one have been removed, the file continues to have a separate **/.inodes** file entry. This is done because the overhead of searching for the directory entry that points to the inode entry would be prohibitive (there are no links from inode entries back to the directory entries).

Removing links

When a file is created, it's given a *link count* of one. As you add and remove links to and from the file, this link count is incremented and decremented.

The disk space occupied by the file data isn't freed and marked as unused in the bitmap until its link count goes to zero *and* all programs using the file have closed it. This allows an open file to remain in use, even though it has been completely unlinked. This behavior is part of that stipulated by POSIX and common Unix practice.

Directory links

Although you can't create hard links to directories, each directory has two hard-coded links already built in:

- **.** ("dot")
- **..** ("dot dot")

The filename "dot" refers to the current directory; "dot dot" refers to the previous (or parent) directory in the hierarchy.

Note that if there's no predecessor, "dot dot" also refers to the current directory. For example, the "dot dot" entry of **/** is simply **/**; you can't go further up the path.

There's no POSIX requirement for a filesystem to include **.** or **..** entries; some filesystems, including flash filesystems and **/dev/shmem**, don't.

Symbolic links

A *symbolic link* (or *symlink*) is a special file that usually has a pathname as its data. When the symbolic link is named in an I/O request—by *open()*, for example—the link portion of the pathname is replaced by the link's "data" and the path is reevaluated.

Symbolic links are a flexible means of pathname indirection and are often used to provide multiple paths to a single file. Unlike hard links, symbolic links can cross filesystems and can also link to directories. You can use the `ln` utility to create a symlink.

In the following example, the directories **/net/node1/usr/fred** and **/net/node2/usr/barney** are linked even though they reside on different filesystems—they're even on different nodes (see the following diagram). You can't do this using hard links, but you can with a symbolic link, as follows:

```
ln -s /net/node2/usr/barney /net/node1/usr/fred
```

Note how the symbolic link and the target directory need not share the same name. In most cases, you use a symbolic link for linking one directory to another directory. However, you can also use symbolic links for files, as in this example:

```
ln -s /net/node1/usr/src/game.c /net/node1/usr/eric/src/sample.c
```

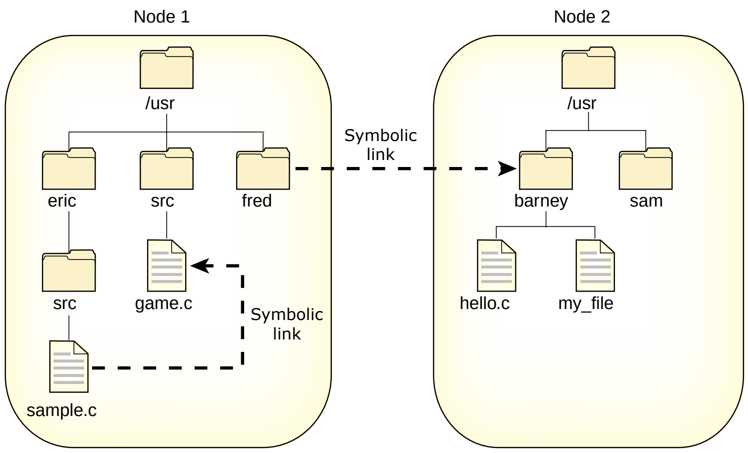


Figure 3: Symbolic links.

Removing a symbolic link deletes only the link, not the target.

Several functions operate directly on the symbolic link. For these functions, the replacement of the symbolic element of the pathname with its target is not performed. These functions include *unlink()* (which removes the symbolic link), *lstat()*, and *readlink()*.

Since symbolic links can point to directories, incorrect configurations can result in problems, such as circular directory links. To recover from circular references, the system imposes a limit on the number of hops; this limit is defined as SYMLOOP_MAX in the **<limits.h>** include file.

Symlinks to symlinks

You can get some surprising results, depending on how you set up the symbolic links in your system. For example:

```
# ln -sP /dev/shmem /some_dir
# echo > /some_dir/my_file
# ln -sP /some_dir/my_file /some_dir/my_link
# ls /some_dir
my_file my_link
# cd /some_dir
# ls
my_file
```

Note that `ls` shows the link if given an explicit path, but otherwise doesn't. Understandably this can cause some confusion and distress. Since it's common for **/tmp** to be a link to **/dev/shmem**, this situation can easily arise for special files created in **/tmp**.

The root of the problem is that when you use *chdir()* or the shell's `cd` command to go to **some_dir**, you actually end up at **/dev/shmem**, because of the **some_dir** symbolic link. But you asked the path manager to create a link under **/some_dir**, not under **/dev/shmem**, and the path manager doesn't care that **/some_dir** is a link somewhere else.

The problem can occur any time a directory symlink exists, where the following special files are created by postfixing the symlink path:

- path manager symlinks, created through *pathmgr_symlink()* or `ln -sP`, as above.
- names attached by a resource manager using *resmgr_attach()* (for example, a Unix domain socket)

We recommend that you always create such links/attachment points by using a canonical path prefix that doesn't contain symlinks. If you do this, then the name will be accessible through the canonical path as well as through the symlink.

Booting

The current boot support is for x86 PC partition-table-based (the same base system as current booting) with a BIOS that supports INT13X (LBA).

The `mkqnx6fs` utility creates a **.boot** directory in the root of the new filesystem. This is always present, and always has an inode of 2 (the root directory itself is inode 1). The `mkqnx6fs` utility also installs a new secondary boot loader in the first 8 KB of the partition (and patches it with the location and offset of the filesystem).

The **fs-qnx6.so** filesystem protects this directory at runtime; in particular it can't be removed or renamed, nor can it exceed 4096 bytes (128 entries). Files placed into the **.boot** directory are assumed to be boot images created with `mkifs`. The name of the file should describe the boot image.

The directory can contain up to 126 entries. You can create other types of object in this directory (e.g., directories or symbolic links) but the boot loader ignores them. The boot loader also ignores certain-sized regular files (e.g., 0 or larger than 2 GB), as well as those with names longer than 27 characters.

The filesystem implicitly suspends snapshots when a boot image is open for writing; this guarantees that the boot loader will never see a partially-written image. You typically build the images elsewhere and then copy them into the directory, and so are open for only a brief time; however this scheme also works if you send the output from `mkifs` directly to the final boot file.

To prevent this from being used as a DOS attack, the default permissions for the boot directory are `root:root rwx------`. You can change the permissions with `chmod` and `chown`, but beware that if you allow everyone to write in this directory, then *anyone* can install custom boot images or delete existing ones.

Snapshots

A *snapshot* is a committed stable view of a Power-Safe filesystem. Each mounted filesystem has one stable snapshot and one working view (in which copy-on-write modifications to the stable snapshot are being made).

Whenever a new snapshot is made, filesystem activity is suspended (to give a stable system), the bitmaps are updated, all dirty blocks are forced to disk, and the alternate filesystem superblock is written (with a higher sequence number). Then filesystem activity is resumed, and another working view is constructed on the old superblock. When a filesystem is remounted after an unclean power failure, it restores the last stable snapshot.

Snapshots are made:

- explicitly, when a global *sync()* of all filesystems is performed
- explicitly, when *fsync()* is called for any file in the Power-Safe filesystem
- explicitly, when switching to read-only mode with `mount -ur`
- periodically, from the timer specified to the `snapshot=` option to the `mount` command (the default is 10 seconds).

You can disable snapshots on a filesystem at a global or local level. When disabled, a new superblock isn't written, and an attempt to make a snapshot fails with an *errno* of EAGAIN (or silently, for the *sync()* or timer cases). If snapshots are still disabled when the filesystem is unmounted (implicitly or at a power failure), any pending modifications are discarded (lost).

Snapshots are also permanently disabled automatically after an unrecoverable error that would result in an inconsistent filesystem. An example is running out of memory for caching bitmap modifications, or a disk I/O error while writing a snapshot. In this case, the filesystem is forced to be read-only, and the current and all future snapshot processing is omitted; the aim being to ensure that the last stable snapshot remains undisturbed and available for reloading at the next mount/startup (i.e., the filesystem always has a guaranteed stable state, even if slightly stale). This is only for certain serious error situations, and generally shouldn't happen.

Manually disabling snapshots can be used to encapsulate a higher-level sequence of operations that must either all succeed or none occur (e.g., should power be lost during this sequence). Possible applications include software updates or filesystem defragmentation.

To disable snapshots at the global level, clear the FS_FLAGS_COMMITTING flag on the filesystem, using the DCMD_FSYS_FILE_FLAGS command to *devctl()*:

```
struct fs_fileflags  flags;

memset( &flags, 0, sizeof(struct fs_fileflags));
flags.mask[FS_FLAGS_GENERIC] = FS_FLAGS_COMMITTING;
flags.bits[FS_FLAGS_GENERIC] = disable ? 0 : FS_FLAGS_COMMITTING;
devctl( fd, DCMD_FSYS_FILE_FLAGS, &flags,
        sizeof(struct fs_fileflags), NULL);
```

This is a single flag for the entire filesystem, and can be set or cleared by any superuser client; thus applications must coordinate the use of this flag among themselves.

Alternatively, you can use the `chattr` utility (as a convenient front-end to the above *devctl()* command):

```
# chattr -snapshot /fs/qnx6
/fs/qnx6: -snapshot
...
# chattr +snapshot /fs/qnx6
/fs/qnx6: +snapshot
```

To disable snapshots at a local level, adjust the QNX6FS_SNAPSHOT_HOLD count on a per-file-descriptor basis, again using the DCMD_FSYS_FILE_FLAGS command to *devctl()*. Each open file has its own hold count, and the sum of all local hold counts is a global hold count that disables snapshots if nonzero. Thus if any client sets a hold count, snapshots are disabled until all clients clear their hold counts.

The hold count is a 32-bit value, and can be incremented more than once (and must be balanced by the appropriate number of decrements). If a file descriptor is closed, or the process terminates, then any local holds it contributed are automatically undone. The advantage of this scheme is that it requires no special coordination between clients; each can encapsulate its own sequence of atomic operations using its independent hold count:

```
struct fs_fileflags  flags;

memset( &flags, 0, sizeof(struct fs_fileflags));
flags.mask[FS_FLAGS_FSYS] = QNX6FS_SNAPSHOT_HOLD;
flags.bits[FS_FLAGS_FSYS] = QNX6FS_SNAPSHOT_HOLD;
devctl( fd, DCMD_FSYS_FILE_FLAGS, &flags,
        sizeof(struct fs_fileflags), NULL);
...
memset( &flags, 0, sizeof(struct fs_fileflags));
flags.mask[FS_FLAGS_FSYS] = QNX6FS_SNAPSHOT_HOLD;
flags.bits[FS_FLAGS_FSYS] = 0;
devctl( fd, DCMD_FSYS_FILE_FLAGS, &flags,
        sizeof(struct fs_fileflags), NULL);
```

In this case, `chattr` isn't particularly useful to manipulate the state, as the hold count is immediately reset once the utility terminates (as its file descriptor is closed). However, it is convenient to report on the current status of the filesystem, as it will display both the global and local flags as separate states:

```
# chattr /fs/qnx6
/fs/qnx6: +snapshot +contiguous +used +hold
```

If `+snapshot` isn't displayed, then snapshots have been disabled via the global flag. If `+hold` is displayed, then snapshots have been disabled due to a global nonzero hold count (by an unspecified number of clients). If `+dirty` is permanently displayed (even after a *sync()*), then either snapshots have been disabled due to a potentially fatal error, or the disk hardware doesn't support full data synchronization (track cache flush).

Enabling snapshots doesn't in itself cause a snapshot to be made; you should do this with an explicit *fsync()* if required. It's often a good idea to *fsync()* both before disabling and after enabling snapshots (the `chattr` utility does this).

DOS filesystem

The DOS filesystem provides transparent access to DOS disks, so you can treat DOS filesystems as though they were QNX Neutrino (POSIX) filesystems. This transparency lets processes operate on DOS files without any special knowledge or work on their part.

The **fs-dos.so** shared object (see the *Utilities Reference*) lets you mount DOS filesystems (FAT12, FAT16, and FAT32) under QNX Neutrino. This shared object is automatically loaded by the `devb-*` drivers when mounting a DOS FAT filesystem.

For information about valid characters for filenames in a DOS filesystem, see the Microsoft Developer Network at *http://msdn.microsoft.com*. FAT 8.3 names are the most limited; they're uppercase letters, digits, and `$%'-_@{}~#()`. VFAT names relax it a bit and add the lowercase letters and `[];,=+`. The QNX Neutrino DOS filesystem silently converts FAT 8.3 filenames to uppercase, to give the illusion that lowercase is allowed (but it doesn't preserve the case).

For more information on the DOS filesystem manager, see **fs-dos.so** in the *Utilities Reference* and Filesystems in the *System Architecture* guide.

Linux Ext2 filesystem

The Linux Ext2 filesystem provided in QNX Neutrino provides transparent access to Linux disk partitions.

Not all Ext2 features are supported, including the following:

- file fragments (subblock allocation)
- large files greater than 2 GB
- filetype extension
- compression
- B-tree directories

The **fs-ext2.so** shared object provides filesystem support for Ext2. This shared object is automatically loaded by the `devb-*` drivers when mounting an Ext2 filesystem.

CAUTION: Although Ext2 is the main filesystem for Linux systems, we don't recommend that you use **fs-ext2.so** as a replacement for the Power-Safe (**fs-qnx6.so**) filesystem. Currently, we don't support booting from Ext2 partitions. Also, the Ext2 filesystem relies heavily on its filesystem checker to maintain integrity; this and other support utilities (e.g., `mke2fs`) aren't currently available for QNX Neutrino.

If an Ext2 filesystem isn't unmounted properly, a filesystem checker is usually responsible for cleaning up the next time the filesystem is mounted. Although the **fs-ext2.so** module is equipped to perform a quick test, it automatically mounts the filesystem as read-only if it detects any significant problems (which should be fixed using a filesystem checker).

This filesystem allows the same characters in a filename as the Power-Safe filesystem; see "*Power-Safe filesystem*," earlier in this chapter.

Flash filesystems

The QNX Neutrino flash filesystem drivers implement a POSIX-compatible filesystem on NOR flash memory devices.

The flash filesystem drivers are standalone executables that contain both the flash filesystem code and the flash device code. There are versions of the flash filesystem driver for different embedded systems hardware as well as PCMCIA memory cards.

Flash filesystems don't include **.** and **..** entries for the current and parent directories.

The naming convention for the drivers is `devf-`*system*, where *system* describes the embedded system. For information about these drivers, see the `devf-*` entries in the *Utilities Reference*.

For more information on the way QNX Neutrino handles flash filesystems, see:

- `mkefs` and `flashctl` in the *Utilities Reference*
- Filesystems in the *System Architecture* guide
- *Building Embedded Systems*

CIFS filesystem

CIFS, the Common Internet File System protocol, lets a client computer perform transparent file access over a network to a Windows system or a Unix system running an SMB server.

It was formerly known as SMB or Server Message Block protocol, which was used to access resources in a controlled fashion over a LAN. File access calls from a client 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 then sends a response back to the client. CIFS runs on top of TCP/IP and uses DNS.

The `fs-cifs` filesystem manager is a CIFS client operating over TCP/IP. To use it, you must have an SMB server and a valid login on that server. The `fs-cifs` utility is primarily intended for use as a client with Windows machines, although it also works with any SMB server, such as OS/2 Peer, LAN Manager, and SAMBA.

The `fs-cifs` filesystem manager requires a TCP/IP transport layer, such as the one provided by `io-pkt*`.

For information about passwords—and some examples—see `fs-cifs` in the *Utilities Reference*.

NFS filesystem

The Network File System (NFS) protocol is a TCP/IP application that supports networked filesystems. It provides transparent access to shared filesystems across networks.

NFS lets a client computer operate on files that reside on a server across a variety of NFS-compliant operating systems. File access calls from a client are converted to NFS protocol (see *RFC 1094* and *RFC 1813*) 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.

In essence, NFS 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.

This filesystem allows the same characters in a filename as the Power-Safe filesystem; see "*Power-Safe filesystem*," earlier in this chapter.

The procedures used in QNX Neutrino for setting up clients and servers may differ from those used in other implementations. To set up clients and servers on a non-QNX Neutrino system, see the vendor's documentation and examine the initialization scripts to see how the various programs are started on that system.

It's actually the clients that do the work required to convert the generalized file access that servers provide into a file access method that's useful to applications and users.

NFS server

An NFS server handles requests from NFS clients that want to access filesystems as NFS mountpoints. For the server to work, you need to start the following programs:

Name:	Purpose:
`rpcbind`	Remote procedure call (RPC) server
`nfsd`	NFS server and `mountd` daemon

The `rpcbind` server maps RPC program/version numbers into TCP and UDP port numbers. Clients can make RPC calls only if `rpcbind` is running on the server.

The `nfsd` daemon reads the **/etc/exports** file, which lists the filesystems that can be exported and optionally specifies which clients those filesystems can be exported to. If no client is specified, any requesting client is given access.

The `nfsd` daemon services both NFS mount requests and NFS requests, as specified by the **exports** file. Upon startup, `nfsd` reads the **/etc/exports.***hostname* file (or, if this file doesn't exist, **/etc/exports**) to determine which mountpoints to service. Changes made to this file don't take affect until you restart `nfsd`.

NFS client

An NFS client requests that a filesystem exported from an NFS server be grafted onto its local namespace. For the client to work, you need to first start the version 2 or 3 of the NFS filesystem manager (`fs-nfs2` or `fs-nfs3`). The file handle in version 2 is a fixed-size array of 32 bytes. With version 3, it's a variable-length array of 64 bytes.

If possible, you should use `fs-nfs3` instead of `fs-nfs2`.

The `fs-nfs2` or `fs-nfs3` filesystem manager is also the NFS 2 or NFS 3 client daemon operating over TCP/IP. To use it, you must have an NFS server and you must be running a TCP/IP transport layer such as that provided by `io-pkt*`. It also needs **socket.so** and **libc.so**.

You can create NFS mountpoints with the `mount` command by specifying `nfs` for the type and `-o ver3` as an option. You must start `fs-nfs3` or `fs-nfs3` *before* creating mountpoints in this manner. If you start `fs-nfs2` or `fs-nfs3` without any arguments, it runs in the background so you can use `mount`.

To make the request, the client uses the `mount` utility, as in the following examples:

- Mount an NFS 2 client filesystem (`fs-nfs2` must be running first):

  ```
  mount -t nfs 10.1.0.22:/home /mnt/home
  ```

- Mount an NFS 3 client filesystem (`fs-nfs3` must be running first):

  ```
  mount -t nfs -o ver3 server_node:/qnx_bin /bin
  ```

In the first example, the client requests that the **/home** directory on an IP host be mounted onto the local namespace as **/mnt/home**. In the second example, NFS protocol version 3 is used for the network filesystem.

Here's another example of a command line that starts and mounts the client:

```
fs-nfs3  10.7.0.197:/home/bob  /homedir
```

Although NFS 2 is older than POSIX, it was designed to emulate Unix filesystem semantics and happens to be relatively close to POSIX.

Universal Disk Format (UDF) filesystem

The Universal Disk Format (UDF) filesystem provides access to recordable media, such as CD, CD-R, CD-RW, and DVD. It's used for DVD video, but can also be used for backups to CD, and so on.

The UDF filesystem is supported by the **fs-udf.so** shared object. The `devb-*` drivers automatically load **fs-udf.so** when mounting a UDF filesystem.

Apple Macintosh HFS and HFS Plus

The Apple Macintosh HFS (Hierarchical File System) and HFS Plus are the filesystems on Apple Macintosh systems.

The **fs-mac.so** shared object provides read-only access to HFS and HFS Plus disks on a QNX Neutrino system. The following variants are recognized: HFS, HFS Plus, HFSJ (HFS Plus with journal), HFS Plus in an HFS wrapper, HFSX (case senstive), and HFS/ISO-9660 hybrid. In QNX Neutrino 7.0.4 or later, you can mount HFSJ filesystems with a hot (or dirty) journal too. In a traditional PC partition table, type 175 is used for HFS.

The `devb-*` drivers automatically load **fs-mac.so** when mounting an HFS or HFS Plus filesystem.

Windows NT filesystem

The NT filesystem is used on Microsoft Windows NT and later. The **fs-nt.so** shared object provides read-only access to NTFS disks on a QNX Neutrino system.

The `devb-*` drivers automatically load **fs-nt.so** when mounting an NT filesystem.

If you want **fs-nt.so** to fabricate **.** and **..** directory entries, specify the `dots=on` option. It doesn't fabricate these entries by default.

Inflator filesystem

QNX Neutrino provides an inflator pass-through filesystem. It's a resource manager that sits in front of other filesystems and decompresses files that were previously compressed by the `deflate` utility.

You typically use `inflator` when the underlying filesystem is a flash filesystem. Using it can almost double the effective size of the flash memory. For more information, see the *Utilities Reference*.

Running more than one pass-through filesystem or resource manager on overlapping pathname spaces might cause deadlocks.

Troubleshooting

Here are some problems that you might have with filesystems:

How can I make a specific flash partition read-only?

Unmount and remount the partition, like this:

```
flashctl -p raw_mountpoint -u
mount -t flash -r raw_mountpoint /mountpoint
```

where *raw_mountpoint* indicates the partition (e.g., **/dev/fs0px**).

How can I determine which drivers are currently running?

1. Use the `find` utility to create a list of pathname mountpoints:

   ```
   find /proc/mount \
   -name '[-0-9]*,[-0-9]*,[-0-9]*,[-0-9]*,[-0-9]*' \
   -prune -print > mountpoints
   ```

2. Use `cut` and `sort` to extract a list of the processes that own the mountpoints:

   ```
   cut -d, -f2 < mountpoints | sort -nu > pidlist
   ```

3. Use `xargs` and `pidin` to display the process ID, long name, and interrupt handlers for each of these processes:

   ```
   xargs -i pidin -p {} -F "%a %n %Q" < pidlist | less
   ```

4. Use `grep` to show the mountpoints for a specified process ID, *pid*:

   ```
   grep pid mountpoints
   ```

5. Use the `-i` option of the `use` utility to show the date of a specified driver, *drivername*:

   ```
   use -i drivername
   ```

This procedure (which approximates the functionality of the Windows XP `driverquery` command) shows the drivers (programs that have mountpoints in the pathname space) that are currently running; it doesn't show those that are merely installed.

Chapter 10
Using Qnet for Transparent Distributed Processing

A QNX Neutrino native network is a group of interconnected computers running only the QNX Neutrino RTOS. In this network, a program can transparently access any resource—whether it's a file, a device, or a process—on any other node (computer) in your local subnetwork. You can even run programs on other nodes.

The Qnet protocol provides transparent networking across a QNX Neutrino network; Qnet implements a local area network that's optimized to provide a fast, seamless interface between QNX Neutrino computers, whatever the type of hardware.

In essence, the Qnet protocol extends interprocess communication (IPC) *transparently* over a network of microkernels—taking advantage of QNX Neutrino's message-passing paradigm to implement native networking.

When you run Qnet, entries for all the nodes in your local subnetwork that are running Qnet appear in the **/net** namespace.

If you run Qnet, anyone else on your network who's running Qnet can examine your files and processes, if the permissions on them allow it. For more information, see:

- "*File ownership and permissions*" in the Working with Files chapter in this guide
- "*Qnet*" in the Securing Your System chapter in this guide
- "Autodiscovery vs static" in the Transparent Distributed Processing Using Qnet chapter of the QNX Neutrino *Programmer's Guide*

For more details, see the Native Networking (Qnet) chapter of the *System Architecture* guide. For information about programming with Qnet, see the Transparent Distributed Networking via Qnet chapter of the *Programmer's Guide*.

When should you use Qnet?

When should you use Qnet, and when TCP/IP or some other protocol? It all depends on what machines you need to connect.

Qnet is intended for a network of trusted machines that are all running QNX Neutrino and that all use the same endian-ness. It lets these machines share all their resources with little overhead. Using Qnet, you can use the QNX 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 requests. Files are protected by the normal permissions that apply to users and groups (see "*File ownership and permissions*" in Working with Files), although you can use Qnet's `maproot` and `mapany` options to control—in a limited way—what others users can do on your machine. Qnet isn't connectionless like NFS; network errors are reported back to the client process.

TCP/IP is intended for more loosely connected machines that can run different operating systems. TCP/IP does authentication to control access to a machine; it's useful for connecting machines that you don't necessarily trust. It's used as the base for specialized protocols such as FTP and Telnet, and can provide high throughput for data streaming. For more information, see the *TCP/IP Networking* chapter in this guide.

NFS was designed for filesystem operations between all hosts, all endians, and is widely supported. It's a connectionless protocol; the server can shut down and be restarted, and the client resumes automatically. It also uses authentication and controls directory access. For more information, see "*NFS filesystem*" in Working with Filesystems.

Conventions for naming nodes

In order to resolve node names, the Qnet protocol follows certain conventions.

node name

A character string that identifies the node you're talking to. This name must be unique in the domain and *can't* contain slashes or periods.

The default node name is the value of the _CS_HOSTNAME configuration string. If your hostname is `localhost` (the default when you first boot), Qnet uses a hostname based on your NIC hardware's MAC address, so that nodes can still communicate.

node domain

A character string that **lsm-qnet.so** adds to the end of the node name. Together, the node name and node domain *must* form a string that's unique for all nodes that are talking to each other. The default is the value of the _CS_DOMAIN configuration string.

***fully qualified node name* (FQNN)**

The string formed by concatenating the node name and node domain. For example, if the node name is **karl** and the node domain name is **qnx.com**, the resulting FQNN is **karl.qnx.com**.

network directory

A directory in the pathname space implemented by **lsm-qnet.so**. Each network directory—there can be more than one on a node—has an associated node domain. The default is **/net**, as used in the examples in this chapter.

> The entries in **/net** for nodes in the same domain as your machine don't include the domain name. For example, if your machine is in the **qnx.com** domain, the entry for **karl** is **/net/karl**; if you're in a different domain, the entry is **/net/karl.qnx.com**.

name resolution

The process by which **lsm-qnet.so** converts an FQNN to a list of destination addresses that the transport layer knows how to get to.

name resolver

A piece of code that implements one method of converting an FQNN to a list of destination addresses. Each network directory has a list of name resolvers that are applied in turn to attempt to resolve the FQNN. The default is the Node Discovery Protocol (NDP).

Software components for Qnet networking

You need the following software entities (along with the hardware) for Qnet networking:

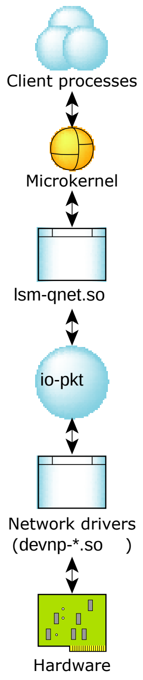


Figure 4: Components of Qnet.

`io-pkt*`

Manager to provide support for dynamically loaded networking modules.

`devnp-*`

Managers that form an interface with the hardware.

lsm-qnet.so

Native network manager to implement Qnet protocols.

Starting Qnet

To start Qnet, load the **lsm-qnet.so** shared object into the network manager.

The `io-pkt*` manager is a process that assumes the central role to load a number of shared objects. It provides the framework for the entire protocol stack and lets data pass between modules. In the case of native networking, the shared objects are **lsm-qnet.so** and networking drivers (**devnp-*.so**). The shared objects are arranged in a hierarchy, with the end user on the top, and hardware on the bottom.

- It's possible to run more than one instance of `io-pkt`, but doing so requires a special setup. If you want to start `io-pkt*` “by hand,” you should `slay` the running `io-pkt*` first.
- You can have at most one instance of Qnet running on a node, even if you're running more than one instance of `io-pkt`.

You can start the `io-pkt*` from the command line, telling it which drivers and protocols to load:

```
$ io-pkt-v4-hc -d abc100  -p qnet
```

This causes `io-pkt-v4-hc` to load the fictitious **devnp-abc100.so** Ethernet driver and the Qnet protocol stack.

Or, you can use the `mount` and ``umount` `` commands to start and stop modules dynamically, like this:

```
$ io-pkt-v6-hc
$ mount -Tio-pkt devnp-abc100.so
$ mount -Tio-pkt lsm-qnet.so
```

To unload the driver, use the `ifconfig destroy` command.

You can't unmount a protocol stack such as TCP/IP or Qnet.

Checking out the neighborhood

Once you've started Qnet, the **/net** directory includes (after a short while—see below) an entry for all other nodes on your local subnetwork that are running Qnet.

You can access files and processes on other machines as if they were on your own computer (at least as far as the permissions allow). For example, to display the contents of a file on another machine, you can use `less`, specifying the path through **/net**:

```
less /net/alonzo/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 alonzo | less
```

You can even run a process on another machine, using your console for input and output, by using the `-f` option to the `on` command:

```
on -f alonzo date
```

Populating /net

When a node boots and starts Qnet along with a network driver, if that node is quiet (i.e., there are no applications on it that try to communicate with other nodes via Qnet), the **/net** directory is slowly populated by the rest of the Qnet nodes, which occasionally broadcast their node information.

The default time interval for this is 30 seconds, and is controlled by the `auto_add=`*X* command-line option to **lsm-qnet.so**. So, 30 seconds after booting, **/net** is probably as full as it's going to get.

You don't have to wait 30 seconds to *talk* to a remote node; immediately after Qnet and the network driver initialize, an application on your node may attempt to communicate with a remote node via Qnet.

When there's an entry in the **/net** directory, all it means is that Qnet now has a mapping from an ASCII text node name to an Ethernet MAC address. It speeds up the node resolution process ever so slightly, and is convenient for people to see what other nodes *might* be on the network.

Entries in **/net** *aren't* deleted until someone tries to use them, and they're found to be invalid.

For example, someone might have booted a node an hour ago, run it for a minute, then shut it down. It will still have an entry in the **/net** directories of the other Qnet nodes, if they never talk to it. If they did talk to it, and establish session connections, everything will eventually be torn down as the session connections time out.

To flush out invalid entries from **/net**, type:

```
ls -l /net &
```

To completely clean out **/net**, type:

```
rmdir /net/*
```

Troubleshooting

All the software components for the Qnet network should work in unison with the hardware to build a native network. If your Qnet network isn't working, you can use various Qnet utilities to fetch diagnostic information to troubleshoot your hardware as well as the network.

Some of the typical questions are:

- *Is Qnet running?*
- *Are `io-pkt*` and the drivers running?*
- *Is the network card functional?*
- *How do I get diagnostic information?*
- *Is the hostname unique?*
- *Are the nodes in the same domain?*

Is Qnet running?

Qnet creates the **/net** directory. Use the following command to make sure that it exists:

```
$ ls /net
```

If you don't see any directory, Qnet isn't running. Ideally, the directory should include at least an entry with the name of your machine (i.e., the output of the `hostname` command). If you're using the Ethernet binding, all other reachable machines are also displayed. For example:

```
joseph/ eileen/
```

Are `io-pkt*` and the drivers running?

As mentioned before, `io-pkt*` is the framework used to connect drivers and protocols. In order to troubleshoot this, use the following `pidin` command:

```
$ pidin -P io-pkt-v4-hc mem
```

Look for the Qnet shared object in the output:

```
   pid tid name                prio STATE           code  data        stack
118802   1 sbin/io-pkt-v4-hc   21o SIGWAITINFO      876K  672K  4096(516K)*
118802   2 sbin/io-pkt-v4-hc   21o RECEIVE          876K  672K  8192(132K)
118802   3 sbin/io-pkt-v4-hc   21r RECEIVE          876K  672K  4096(132K)
118802   4 sbin/io-pkt-v4-hc   21o RECEIVE          876K  672K  4096(132K)
118802   5 sbin/io-pkt-v4-hc   20o RECEIVE          876K  672K  4096(132K)
118802   6 sbin/io-pkt-v4-hc   10o RECEIVE          876K  672K  4096(132K)
          libc.so.2          @b0300000             436K   12K
          devnp-abc100.so    @b8208000              40K  4096
          lsm-qnet.so        @b8213000             168K   36K
```

If the output includes an **lsm-qnet.so** shared object, Qnet is running.

Is the network card functional?

To determine whether or not the network card is functional, i.e., transmitting and receiving packets, use the `nicinfo` command.

If you're logged in as **root**, your *PATH* includes the directory that contains the `nicinfo` executable; if you're logged in as another user, you have to specify the full path:

```
$ /usr/sbin/nicinfo
```

Now figure out the diagnostic information from the following output:

```
en0:
  AMD PCNET-32 Ethernet Controller

  Physical Node ID .......................... 000C29 DD3528
  Current Physical Node ID .................. 000C29 DD3528
  Current Operation Rate .................... 10.00 Mb/s
  Active Interface Type ..................... UTP
  Maximum Transmittable data Unit ........... 1514
  Maximum Receivable data Unit .............. 1514
  Hardware Interrupt ........................ 0x9
  I/O Aperture .............................. 0x1080 - 0x10ff
  Memory Aperture ........................... 0x0
  Promiscuous Mode .......................... Off
  Multicast Support ......................... Enabled

  Packets Transmitted OK .................... 588
  Bytes Transmitted OK ...................... 103721
  Memory Allocation Failures on Transmit ..... 0

  Packets Received OK ....................... 11639
  Bytes Received OK ......................... 934712
  Memory Allocation Failures on Receive ...... 0

  Single Collisions on Transmit ............. 0
  Deferred Transmits ........................ 0
  Late Collision on Transmit errors ......... 0
  Transmits aborted (excessive collisions) ... 0
  Transmit Underruns ........................ 0
  No Carrier on Transmit .................... 0
  Receive Alignment errors .................. 0
  Received packets with CRC errors .......... 0
  Packets Dropped on receive ................ 0
```

You should take special note of the `Packets Transmitted OK` and `Packets Received OK` counters. If they're zero, the driver might not be working, or the network might not be connected. Verify that the driver has correctly auto-detected the `Current Operation Rate`.

How do I get diagnostic information?

You can find diagnostic information in **/proc/qnetstats**. If this file doesn't exist, Qnet isn't running.

The **qnetstats** file contains a lot of diagnostic information that's meaningful to a Qnet developer, but not to you. However, you can use `grep` to extract certain fields:

```
# cat /proc/qnetstats | grep "compiled"
**** Qnet compiled on Jun  3 2008 at 14:08:23 running on EAdd3528
```

or:

```
# cat /proc/qnetstats | grep -e "ok" -e "bad"
  txd ok       930
  txd bad      0
  rxd ok       2027
  rxd bad dr   0
  rxd bad L4   0
```

If you need help getting Qnet running, our Technical Support department might ask you for this information.

Is the hostname unique?

Use the `hostname` command to see the hostname. This hostname must be unique for Qnet to work.

Are the nodes in the same domain?

If the nodes aren't in the same domain, you have to specify the domain. For example:

```
ls /net/kenneth.qnx.com
```

Chapter 11 TCP/IP Networking

The term *TCP/IP* implies two distinct protocols: TCP and IP. Since these protocols have been used so commonly together, TCP/IP has become a standard terminology in today's Internet. Essentially, TCP/IP refers to network communications where the TCP transport is used to deliver data across IP networks.

This chapter provides information on setting up TCP/IP networking on a QNX Neutrino network. It also provides troubleshooting and other relevant details from a system-administration point of view. A QNX Neutrino-based TCP/IP network can access resources located on any other system that supports TCP/IP.

Overview of TCP/IP

Let's start with some definitions.

Clients and servers

There are two types of TCP/IP hosts: clients and servers. A client requests TCP/IP service; a server provides it. In planning your network, you must decide which hosts will be servers and which will be clients.

For example, if you want to `telnet` *from* a machine, you need to set it up as a client; if you want to `telnet` *to* a machine, it has to be a server.

Hosts and gateways

In TCP/IP terminology, we always refer to network-accessible computers as either *hosts* or *gateways*.

Host

A node running TCP/IP that doesn't forward IP packets to other TCP/IP networks; a host usually has a single interface (network card) and is the destination or source of TCP/IP packets.

Gateway

A node running TCP/IP that forwards IP packets to other TCP/IP networks, as determined by its routing table. These systems have two or more network interfaces. If a TCP/IP host has Internet access, there must be a gateway located on its network.

In order to use TCP/IP, you need an IP address, and you also need the IP address of the host you wish to communicate with. You typically refer to the remote host by using a textual name that's resolved into an IP address by using a name server.

Name servers

A *name server* is a database that contains the names and IP addresses of hosts. You normally access a TCP/IP or Internet host with a textual name (e.g., **www.qnx.com**) and use some mechanism to translate the name into an IP address (e.g., 209.226.137.1).

The simplest way to do this mapping is to use a table in the **/etc/hosts** file. This works well for small to medium networks; if you have something a bit more complicated than a small internal network with a few hosts, you need a name server (e.g., for an ISP connection to the Internet).

When you use a name to connect to a TCP/IP host, the name server is asked for the corresponding IP address, and the connection is then made to that IP address. You can use either:

- a name server entry in the configuration string _CS_RESOLVE

  or:

- a name server entry in the **/etc/resolv.conf** file. For example:

```
nameserver 10.0.0.2
nameserver 10.0.0.3
```

For more information on finding TCP/IP hostnames and name servers, see **/etc/hosts**, **/etc/nsswitch.conf** and **/etc/resolv.conf** in the *Utilities Reference*.

If the name server isn't responding, there's a timeout of 1.5 minutes per name server. You can't change this timeout, but many TCP/IP utilities have a `-n` option that you can use to prevent name lookups.

Routing

Routing determines how to get a packet to its intended destination. The general categories of routing are:

Minimal routing

You will only be communicating with hosts on your own network. For example, you're isolated on your own network.

Static routing

If you're on a network with a small (and static over time) number of gateways, then you can use the `route` command to manually manipulate the TCP/IP routing tables and leave them that way.

This is a very common configuration. If a host has access to the Internet, it likely added one static route called a *default route*. This route directs all the TCP/IP packets from your host that aren't destined for a host on your local network to a gateway that provides access to the Internet.

Dynamic routing

If you're on a network with more than one possible route to the same destination on your network, you might need to use dynamic routing. This relies on routing protocols to distribute information about the changing state of the network. If you need to react to these changes, run `routed`, which implements the Routing Information Protocol (RIP) and RIPv2.

There's often confusion between routing and routing protocols. The TCP/IP stack determines the routing by using routing tables; routing protocols let those tables change.

Software components for TCP/IP networking

To use TCP/IP, you need the following software components:

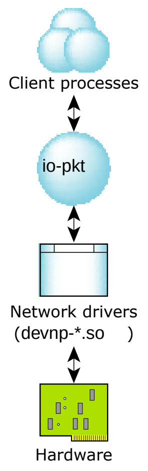


Figure 5: Components of TCP/IP in QNX Neutrino.

`io-pkt*`

Manager that provides support for dynamically loaded networking modules. It includes a fully featured TCP/IP stack derived from the NetBSD code base.

`devnp-*`

Managers that form an interface with the hardware.

To set configuration parameters, use the `ifconfig` and `route` utilities, as described below.

If you're using the Dynamic Host Configuration Protocol (DHCP), you can use `dhclient` to set the configuration parameters for you as provided by the DHCP server.

The TCP/IP stack is based on the NetBSD TCP/IP stack, and it supports similar features. To configure the stack, use the `ifconfig` and `route` utilities as described below.

To configure an interface with an IP address, you must use the `ifconfig` utility. To configure your network interface with an IP address of **10.0.0.100**, you would use the following command:

`ifconfig` *if_name* `10.0.0.100`

where *if_name* is the interface name that the driver uses.

If you also want to specify your gateway, use the `route` command:

```
route add default 10.0.0.1
```

This configures the gateway host as **10.0.0.1**.

If you then want to view your network configuration, use the `netstat` command (`netstat -in` displays information about the network interfaces):

```
Name Mtu    Network   Address           Ipkts Ierrs Opkts Oerrs Coll
lo0  32976  <Link>                       0     0     0     0     0
lo0  32976  127       127.0.0.1          0     0     0     0     0
```

```
en0  1500   <Link>    00:50:da:c8:61:92 21    0     2     0     0
en0  1500   10        10.0.0.100        21    0     2     0     0
```

To display information about the routing table, use `netstat -rn`; the resulting display looks like this:

```
Routing tables

Internet:
Destination Gateway    Flags Refs Use Mtu Interface
default     10.0.0.1   UGS   0    0   -   en0
10          10.0.0.100 U     1    0   -   en0
10.0.0.100  10.0.0.100 UH    0    0   -   lo0
127.0.0.1   127.0.0.1  UH    0    0   -   lo0
```

The table shows that the default route to the gateway was configured (10.0.0.1).

Running the Internet daemons

If a host is a server, it invokes the appropriate daemon to satisfy a client's requests. A TCP/IP server typically runs the `inetd` daemon, also known as the Internet super-server.

CAUTION: Running `inetd` lets outside users try to connect to your machine and thus is a potential security issue if you don't configure it properly.

The `inetd` daemon listens for connections on some well-known ports, as defined in **/etc/inetd.conf**, in the TCP/IP network. On receiving a request, it runs the corresponding server daemon. For example, if a client requests a remote login by invoking `rlogin`, then `inetd` starts `rlogind` (remote login daemon) to satisfy the request. In most instances, responses to client requests are handled this way.

You use the super-server configuration file **/etc/inetd.conf** to specify the daemons that `inetd` can start.

As shipped in the QNX Neutrino distribution, the file contains commented-out descriptions of all currently shipped QNX Neutrino TCP/IP daemons and some nonstandard `pidin` services. You need to edit **inetd.conf** and uncomment the descriptions of the ones you want to use.

When it starts, `inetd` reads its configuration information from this configuration file. It includes these commonly used daemons:

`ftpd`

File transfer.

`rlogind`

Remote login.

`rshd`

Remote shell.

`telnetd`

Remote terminal session.

`tftpd`

DARPA trivial file transfer.

- Remember that you shouldn't manually start the daemon processes listed in this file; they expect to be started by `inetd`.
- Running `rshd` or `rlogind` can open up your machine to the world. Use the **/etc/hosts.equiv** or **~/.rhosts** files (or both) to identify trusted users, but be very careful.

You may also find other resident daemons that can run independently of `inetd`—see the *Utilities Reference* for descriptions:

`bootpd`

Internet boot protocol server.

`dhcpd`

Dynamic Host Configuration Protocol daemon.

`mrouted`

Distance-Vector Multicast Routing Protocol (DVMRP) daemon.

`named`

Internet domain name server

`ntpd`

Network Time Protocol daemon.

`routed`

RIP and RIPv2 routing protocol daemon

`rwhod`

System status database.

`nfsd`

NFS server.

These daemons listen on their own TCP ports and manage their own transactions. They usually start when the computer boots and then run continuously, although to conserve system resources, you can have `inetd` start `bootpd` only when a boot request arrives.

Running multiple instances of the TCP/IP stack

If your system has more than one Network Interface Card, you may need to run multiple instances of the TCP/IP stack.

If the NICs are the same type, you have to specify their PCI indexes (which you can determine by using the `pci-tool` command) when you start `io-pkt`. You can use stack instance numbers and prefixes to identify the instances of the stack. For example, let's start two stacks using the fictitious **devnp-abc100** driver:

```
io-pkt-v4-hc -d abc100 pci=0x0
io-pkt-v4-hc -i2 -d abc100 pci=0x1 -ptcpip prefix=/sock2
```

In the second command line:

- The `-i` option tells `io-pkt-v4-hc` to use a stack instance number of 2 and to register itself as **io-pkt2**.
- The `prefix` option causes the stack to be registered as **/sock2/dev/socket** instead of the default, **/dev/socket**.

TCP/IP applications that wish to use the second stack must specify the *SOCK* environment variable. If you don't specify *SOCK*, the command uses the first TCP/IP stack. For example:

- `SOCK=/sock2 telnet 10.59`
- `SOCK=/sock2 netstat -in`
- `SOCK=/sock2 ifconfig` *if_name* `192.168.2.10` (where *if_name* is the interface name that the driver uses)
- `SOCK=/sock2 pfctl -p /sock2/dev/pf...` (Note that for `pfctl` you also need to use the `-p` option to identify the device.)

If you're using `mount`, you can add the stack instance number to the name of the manager. For example, to load **lsm-pf-v4.so** into the second instance of the stack, type:

```
mount -Tio-pkt2 lsm-pf-v4.so
```

Dynamically assigned TCP/IP parameters

When you add a host to the network or connect your host to the Internet, you need to assign an IP address to your host and set some other configuration parameters.

There are a few common mechanisms for doing this:

- Dial-up providers use the Point-to-Point Protocol (PPP).
- Broadband providers, such as Digital Subscriber Line (DSL) or Cable, use Point-to-Point Protocol over Ethernet (PPPoE) or DHCP.
- A typical corporate network deploys DHCP.

Along with your IP address, the servers implementing these protocols can supply your gateway, netmask, name servers, and even your printer in the case of a corporate network. Users don't need to manually configure their host to use the network.

QNX Neutrino also implements another autoconfiguration protocol called AutoIP (zeroconf IETF draft). This autoconfiguration protocol is used to assign link-local IP addresses to hosts in a small network. It uses a peer-negotiation scheme to determine the link-local IP address to use instead of relying on a central server.

Using PPPoE

PPPoE stands for Point-to-Point Protocol over Ethernet. It's a method of encapsulating your data for transmission over a bridged Ethernet topology.

PPPoE is a specification for connecting users on an Ethernet network to the Internet through a broadband connection, such as a single DSL line, wireless device, or cable modem. Using PPPoE and a broadband modem, LAN users can gain individual authenticated access to high-speed data networks.

By combining Ethernet and the Point-to-Point Protocol (PPP), PPPoE provides an efficient way to create a separate connection to a remote server for each user. Access, billing, and choice of service are managed on a per-user basis, rather than a per-site basis. It has the advantage that neither the telephone company nor the Internet service provider (ISP) needs to provide any special support.

Unlike dialup connections, DSL and cable modem connections are always on. Since a number of different users are sharing the same physical connection to the remote service provider, a way is needed to keep track of which user traffic should go to where, and which user should be billed. PPPoE lets each user-remote site session learn each other's network addresses (during an initial exchange called *discovery*). Once a session is established between an individual user and the remote site (for example, an Internet service provider), the session can be monitored for billing purposes. Many apartment houses, hotels, and corporations are now providing shared Internet access over DSL lines using Ethernet and PPPoE.

A PPPoE connection is composed of a client and a server. Both the client and server work over any Ethernet-like interface. It's used to hand out IP addresses to the clients, based on the user (and computer if desired), as opposed to computer-only authentication. The PPPoE server creates a point-to-point connection for each client.

Establishing a PPPoE session

The io-pkt-* stack provides PPP-to-Ethernet services. Start io-pkt* with the appropriate driver. For example (with the fictitious ABC100 driver):

```
io-pkt-v6-hc -d abc100
```

Starting a point-to-point connection over PPPoE session

Use pppoectl to configure and initiate the PPPoE session. Here's an example of the commands you use to bring up a PPPoE connection:

1. Make sure the Ethernet interface is up (or else it won't send any packets):

   ```
   ifconfig ne0 up
   ```

2. Let pppoe0 use ne0 as its Ethernet interface:

   ```
   pppoectl -e ne0 pppoe0
   ```

3. Configure authentication:

   ```
   pppoectl pppoe0 \
      myauthproto=pap \
      myauthname=XXXXX \
      myauthsecret=YYYYY \
      hisauthproto=none
   ```

4. Configure the pppoe0 interface itself. These addresses are magic, meaning we don't care about either address, so we let the remote PPP choose them:

   ```
   ifconfig pppoe0 0.0.0.0 0.0.0.1 netmask 0xffffffff up
   ```

You can use ifwatchd to spawn scripts when the IP address is configured.

If PPPoE has problems connecting to certain sites on the Internet, see PPPOE and Path MTU Discovery in the QNX Neutrino technotes.

Using DHCP

A TCP/IP host uses the DHCP (Dynamic Host Configuration Protocol) to obtain its configuration parameters (IP address, gateway, name servers, and so on) from a DHCP server that contains the configuration parameters of all the hosts on the network.

The QNX Neutrino DHCP client, dhclient, obtains these parameters and configures your host for you to use the Internet or local network.

If your DHCP server supplies options (configuration parameters) that dhclient doesn't know how to apply, dhclient passes them to a script that it executes. You can use this script to apply any options you want to use outside of those that dhclient sets for you. For more information, see the entry for dhclient in the *Utilities Reference*.

Using AutoIP

AutoIP is a module that you must mount into `io-pkt*`. It's used for quick configuration of hosts on a small network.

AutoIP assigns a link-local IP address from the 169.254/16 network to its interface if no other host is using this address. The advantage of using AutoIP is that you don't need a central configuration server. The hosts negotiate among themselves which IP addresses are free to use, and monitor for conflicts.

It's common to have a host employ both DHCP and AutoIP at the same time. When the host is first connected to the network, it doesn't know if a DHCP server is present or not. If you use `dhclient` to create an alias, then both a link-local IP address and DHCP IP address can be assigned to your interface at the same time. If the DHCP server isn't present, `dhclient` times out, leaving the link-local IP address active. If a DHCP server becomes available later, `dhclient` can be restarted and a DHCP IP address applied without interfering with any TCP/IP connections currently using the link-local IP address.

Having both a DHCP-assigned address and a link-local address active at the same time lets you communicate with hosts that have link-local IP addresses and those that have regular IP addresses. For more information, see `autoipd` and `dhclient` in the *Utilities Reference*.

Troubleshooting

If you're having trouble with your TCP/IP network (e.g., you can't send packets over the network), you need to use several utilities for troubleshooting. These utilities query hosts, servers, and the gateways to fetch diagnostic information to locate faults.

Some of the typical queries are:

- *Are `io-pkt*` and the drivers running?*
- *What is the name server information?*
- *How do I map hostnames to IP addresses?*
- *How do I get the network status?*
- *How do I make sure I'm connected to other hosts?*
- *How do I display information about an interface controller?*

Are `io-pkt*` and the drivers running?

As mentioned before, `io-pkt*` is the framework used to connect drivers and protocols. In order to troubleshoot this, use the `pidin` command:

```
$ pidin -P io-pkt-v4-hc mem
```

The output should be something like this:

```
     pid tid name                prio STATE            code  data        stack
  126996   1 sbin/io-pkt-v4-hc   21o SIGWAITINFO      872K  904K  8192(516K)*
  126996   2 sbin/io-pkt-v4-hc   21o RECEIVE          872K  904K  8192(132K)
  126996   3 sbin/io-pkt-v4-hc   21r RECEIVE          872K  904K  4096(132K)
  126996   4 sbin/io-pkt-v4-hc   21o RECEIVE          872K  904K  4096(132K)
  126996   5 sbin/io-pkt-v4-hc   20o RECEIVE          872K  904K  4096(132K)
  126996   6 sbin/io-pkt-v4-hc    9o RECEIVE          872K  904K  4096(132K)
            libc.so.4          @b0300000             444K   16K
            devnp-abc100.so    @b8209000              40K  4096
            lsm-qnet.so        @b8214000             168K   36K
```

You should see a shared object for a network driver (in this case the fictitious **devnp-abc100.so**). You can also use the `pidin ar` and `ifconfig` commands to get more information about how the networking is configured.

What is the name server information?

Use the following command to get the name server information:

```
getconf _CS_RESOLVE
```

If you aren't using the configuration string, type:

```
cat /etc/resolv.conf
```

How do I map hostnames to IP addresses?

The **/etc/hosts** file contains information regarding the known hosts on the network.

For each host, a single line should be present with the following information:

internet_address *official_host_name* *aliases*

Display this file by using the following command:

```
cat /etc/hosts
```

How do I get the network status?

Use the following `netstat` commands to get the network status:

`netstat -in`

List the interfaces, including the MAC and IP addresses that they've been configured with.

`netstat -rn`

Display the network routing tables that determine how the stack can reach another host. If there's no route to another host, you get a “no route to host” error.

`netstat -an`

List information about TCP/IP connections to or from your system. This includes the state of the connections or the amount of data pending on the connections. It also provides the IP addresses and ports of the local and remote ends of the connections.

For more information about `netstat`, see the *Utilities Reference*.

How do I make sure I'm connected to other hosts?

Use the `ping` utility to determine if you're connected to other hosts.

For example:

```
ping isp.com
```

On success, `ping` displays something like this:

```
PING isp.com (10.0.0.1): 56 data bytes
64 bytes from 10.0.0.1: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 10.0.0.1: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=255 time=0 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=255 time=0 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=255 time=0 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=255 time=0 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=255 time=0 ms
```

This report continues until you terminate `ping`, for example, by pressing **Ctrl–C**.

How do I display information about an interface controller?

Use the `nicinfo` command:

```
/usr/sbin/nicinfo device
```

If you aren't logged in as **root**, you have to specify the full path to `nicinfo`.

This utility displays information about the given network interface connection, or all interfaces if you don't specify one. The information includes the number of packets transmitted and received, collisions, and other errors, as follows:

```
3COM (90xC) 10BASE-T/100BASE-TX Ethernet Controller
  Physical Node ID ................. 000103 E8433F
  Current Physical Node ID ......... 000103 E8433F
  Media Rate ....................... 10.00 Mb/s half-duplex UTP
  MTU .............................. 1514
  Lan .............................. 0
  I/O Port Range ................... 0xA800 -> 0xA87F
  Hardware Interrupt ............... 0x7
  Promiscuous ...................... Disabled
  Multicast ........................ Enabled

  Total Packets Txd OK ............. 1585370
  Total Packets Txd Bad ............ 9
  Total Packets Rxd OK ............. 11492102
  Total Rx Errors .................. 0

  Total Bytes Txd .................. 102023380
  Total Bytes Rxd .................. 2252658488

  Tx Collision Errors .............. 39598
  Tx Collisions Errors (aborted) ... 0
  Carrier Sense Lost on Tx ......... 0
  FIFO Underruns During Tx ......... 0
  Tx deferred ...................... 99673
  Out of Window Collisions ......... 0
  FIFO Overruns During Rx .......... 0
  Alignment errors ................. 0
  CRC errors ....................... 0.
```

Chapter 12
Backing Up Data

No matter how reliable your hardware and electrical supply are, or how sure you are that you'll never accidentally erase all your work, it's just common sense to keep backups of your files. Backup strategies differ in ease of use, speed, robustness, and cost.

Although we'll discuss different types of archives below, here's a quick summary of the file extensions associated with the different utilities:

Extension	Utility
.tar	`pax` or `tar`
.cpio	`pax` or `cpio`
.gz	`gzip` or `gunzip`
.tar.gz or **.tgz**	`tar -z`
.z or **.F**	`melt`

No matter how robust a filesystem is designed to be, there will always be situations in the real world where disk corruption will occur. Hardware will fail eventually, power will be interrupted, and so on.

The Power-Safe filesystem is designed so that it should never be corrupted; you'll always have a complete version of its data. For more information, see "Power-Safe filesystem" in the Filesystems chapter of the *System Architecture* guide. However, it's still a good idea to back up your data.

Backup strategies

Your backup strategy will consist of making one or more backups on a periodic or triggered basis.

For each backup you incorporate in your strategy, you have to choose:

- the storage media and location of the backup data
- how to archive, and optionally, compress your data
- the contents, and frequency or trigger condition of the backup
- automated versus manual backup
- local versus remote control of the backup

Often, a comprehensive backup strategy incorporates some backups on the local side (i.e., controlled and stored on the same machine that the data is located on), and others that copy data to a remote machine. For example, you might automatically back up a developer's data to a second hard drive partition on a daily basis and have a central server automatically back up the developer's data to a central location on a weekly basis.

Choosing backup storage media and location

Early in the process of determining your backup strategy, you're likely to choose the location of your data backups and the media to store the backups on, because these choices are the primary factors that affect the hardware and media costs associated with the system.

To make the best choice, first take a close look at *what* you need to back up, and *how often* you need to do it. This information determines the storage capacity, transfer bandwidth, and the degree to which multiple users can share the resource.

Your choices of backup media vary, depending on whether you create backup copies of your data on a local machine or on a remote machine by transferring the data via a network:

- Local backups offer the advantage of speed and potentially greater control by the end user, but are limited to backup technologies and media types that QNX Neutrino supports directly.
- Remote backups often allow use of company-wide backup facilities and open up additional storage options, but are limited by the need to transfer data across a network and by the fact that the facilities are often shared, restricting your access for storing or retrieving your backups.

Choosing a backup format

When backing up your data, you need to decide whether to back up each file and directory separately, or in an archive with a collection of other files. You also need to decide whether or not to compress your data to reduce the storage requirements for your backups.

The time lost to compression and decompression may be offset to a degree by the reduced time it takes to write or read the compressed data to media or to transfer it through a network. To reduce the expense of compression, you may choose to compress the backup copies of your data as a background task after the data has been copied—possibly days or weeks after—to reduce the storage requirements of older backups while keeping newer backups as accessible as possible.

Controlling your backup

You should back up often enough so that you can recover data that's still current or can be made current with minimal work.

In a software development group, this may range from a day to a week. Each day of out-of-date backup will generally cost you a day of redevelopment. If you're saving financial or point-of-sale data, then daily or even twice-daily backups are common. It's a good idea to maintain off-site storage.

Chapter 13 Securing Your System

Now that more and more computers and other devices are hooked up to insecure networks, security has become a very important issue. The word “security” can have many meanings, but in a computer context, it generally means preventing unauthorized people from making your computer do things that you don't want it to do.

There are vast tracts of security information in books and on the Internet. This chapter provides a very brief introduction to the subject of security, points you toward outside information and resources, and discusses security issues that are unique to QNX Neutrino.

For additional information, see the Security Developer's Guide.

General OS security

It should be fairly obvious that security is important; you don't want someone to take control of a device and disrupt its normal functioning—imagine the havoc if someone could stop air traffic control systems or hospital equipment from functioning properly.

The importance of security to an individual machine depends on the context:

- A machine behind a strong firewall is less vulnerable than one connected to a public network.
- One that doesn't even have a network connection is in even less danger.

Part of securing a machine is identifying the level of risk. By classifying threats into categories, we can break down the issues and see which ones we need to concern ourselves with.

Remote and local attacks

We can break the broad division of security threats, also known as *exploits*, into categories:

Remote exploit

The attacker connects to the machine via the network and takes advantage of bugs or weaknesses in the system.

Local attack

The attacker has an account on the system in question and can use that account to attempt unauthorized tasks.

Remote exploits

Remote exploits are generally much more serious than local ones, but fortunately, remote exploits are much easier to prevent and are generally less common.

For example, suppose you're running `bind` (a DNS resolver) on port 53 of a publicly connected computer, and the particular version has a vulnerability whereby an attacker can send a badly formed query that causes `bind` to open up a shell that runs as **root** on a different port of the machine. An attacker can use this weakness to connect to and effectively “own” the computer.

This type of exploit is often called a *buffer overrun* or *stack-smashing* attack and is described in the article, *Smashing the Stack for Fun and Profit* by Aleph One (see *http://www.insecure.org/stf/smashstack.txt*). The simple solution to these problems is to make sure that you know which servers are listening on which ports, and that you're running the latest versions of the software. If a machine is publicly connected, don't run any more services than necessary on it.

Local exploits

Local exploits are much more common and difficult to prevent.

Having a local account implies a certain amount of trust, and it isn't always easy to imagine just how that trust could be violated. Most local exploits involve some sort of elevation of privilege, such as turning a normal user into the superuser, **root**.

Many local attacks take advantage of a misconfigured system (e.g., file permissions that are set incorrectly) or a buffer overrun on a binary that's set to run as **root** (known as a setuid binary). In the embedded world—where QNX Neutrino is often used—local users aren't as much of an issue and, in fact, many systems don't even have a shell shipped with them.

Effects of attacks

Another way of classifying exploits is by their effect.

Takeover attacks

These let the user take the machine over, or at least cause it to do something unpredictable to the owner but predictable to the attacker.

Denial Of Service (DOS) attacks

These are just disruptions. An example of this is flood-pinging a machine to slow down its networking to the point that it's unusable. DOS attacks are notoriously difficult to deal with, and often must be handled in a reactive rather than proactive fashion.

As an example, there are very few systems that can't be brought to their knees by a malicious local user although, with such tools as the `ksh`'s `ulimit` builtin command, you can often minimize these attacks.

Using these divisions, you can look at a system and see which classes of attacks it could potentially be vulnerable to, and take steps to prevent them.

Viruses

A virus is generally considered to be an infection that runs code on the host (e.g., a Trojan horse). Viruses need an entry point and a host.

The entry points for a virus include:

- an open interface (e.g., ActiveX)—QNX Neutrino has none
- a security hole (such as buffer overflows)—these are specific to flaws in specific services, based on a common industry-standard code base. These are limited, since we ship only a limited set of standard (BSD) services.

The hosts for a virus are system-call interfaces that are accessible from the point of entry (an infected program), such as `sendmail` or an HTTP server. The hosts are platform-specific, so a virus for Linux would in all likelihood terminate the host under QNX Neutrino as soon as it tried to do anything damaging.

The viruses that circulate via email are OS-specific, generally targeted at Windows, and can't harm QNX Neutrino systems, since they simply aren't compatible. Most Unix-style systems aren't susceptible to viruses since the ability to do (much) damage is limited by the host. We have never heard of a true virus that could infect QNX Neutrino.

In addition, since deployed QNX Neutrino systems are highly customized to their designated application, they often don't contain the software that's open to such attacks (e.g., logins, web browsers, email, Telnet and FTP servers).

QNX Neutrino security in general

QNX Neutrino is a Unix-style operating system, so almost all of the general Unix security information (whether generic, Linux, BSD, etc.) applies to QNX Neutrino as well. A quick Internet search for Unix or Linux security will yield plenty of papers. You'll also find many titles at a bookstore or library.

We don't market QNX Neutrino as being either more or less secure than other operating systems in its class. That is, we don't attempt to gain a security certification such as is required for certain specialized applications. However, we do conduct internal security audits of vulnerable programs to correct potential exploits.

For flexibility and familiarity, QNX Neutrino uses the generic Unix security model of user accounts and file permissions, which is generally sufficient for all our customers. In the embedded space, it's fairly easy to lock down a system to any degree without compromising operation. The ultrasecure systems that need certifications are generally servers, as opposed to embedded devices.

For more information, see *Managing User Accounts*, and "*File ownership and permissions*" in Working with Files.

QNX Neutrino-specific security issues

As the above section notes, QNX Neutrino is potentially vulnerable to most of the same threats that other Unix-style systems face. In addition, there are also some issues that are unique to QNX Neutrino.

Message passing

Our basic model of operation relies on message passing between the OS kernel, process manager and other services.

There are potential local exploits in that area that wouldn't exist in a system where all drivers live in the same address space as the kernel. Of course, the potential weakness is outweighed by the demonstrated strength of this model, since embedded systems generally aren't overly concerned with local attacks.

Security policy and mandatory access controls dictate which processes can communicate with what other processes. Even if there is a security weakness in some driver and an attacker is able to gain access to the system, if they are unable to communicate with the driver, it will difficult to profit from the weakness.

For more information about the microkernel design and message passing, see the QNX Neutrino Microkernel and Interprocess Communication (IPC) chapters of the *System Architecture* guide.

`pdebug`

Our remote debug agent, `pdebug`, runs on a target system and communicates with the `gdb` debugger on the host.

The `pdebug` agent can run as a dedicated server on a port, be spawned from `inetd` with incoming connections, or be spawned by `qconn`.

The `pdebug` agent is generally run as **root**, so anyone can upload, download, or execute any arbitrary code at **root**'s privilege level. This agent was designed to be run on development systems, not production machines. There's no means of authentication or security, and none is planned for the future. See the section on `qconn` below.

`qconn`

The `qconn` daemon is a server that runs on a target system and handles all incoming requests from our IDE.

The `qconn` server spawns `pdebug` for debugging requests, profiles applications, gathers system information, and so on.

Like `pdebug`, `qconn` is inherently insecure and is meant for development systems.

Qnet

Qnet is QNX Neutrino's transparent networking protocol.

It's described in the *Using Qnet for Transparent Distributed Processing* chapter in this guide, and in the Native Networking (Qnet) chapter of the *System Architecture* guide.

Qnet displays other QNX Neutrino machines on the network in the filesystem and lets you treat remote systems as extensions of the local machine. It does no authentication beyond getting a user ID from the incoming connection, so be careful when running it on a machine that's accessible to public networks.

Security policy offers some control over what processes on one node can do to another. When you are securing a node in a Qnet network, choose a policy that prohibits all Qnet access so that the secured node can use Qnet to communicate with other nodes and other nodes cannot connect to the secured one. Limit processes that can be connected to, over Qnet, to ones that are considered low risk.

To make Qnet more secure, you can use the `maproot` and `mapany` options, which map incoming connections (**root** or anyone, respectively) to a specific user ID. For more information, see **lsm-qnet.so** in the *Utilities Reference*.

IPSec

IPsec is a security protocol for the Internet Protocol layer that you can use, for example, to set up a secure tunnel between machines or networks.

It consists of these subprotocols:

AH (Authentication Header)

Guarantees the integrity of the IP packet and protects it from intermediate alteration or impersonation, by attaching a cryptographic checksum computed by one-way hash functions.

ESP (Encapsulated Security Payload)

Protects the IP payload from wire-tapping, by encrypting it using secret-key cryptography algorithms.

IPsec has these modes of operation:

Transport

Protects peer-to-peer communication between end nodes.

Tunnel

Supports IP-in-IP encapsulation operation and is designed for security gateways, such as VPN configurations.

The IPsec support is subject to change as the IPsec protocols develop.

For more information, see IPSec in the QNX Neutrino *C Library Reference*.

Setting up a firewall

Just as a building or vehicle uses specially constructed walls to prevent the spread of fire, so computer systems use *firewalls* to prevent or limit access to certain applications or systems and to protect systems from malicious attacks.

To create a firewall under QNX Neutrino, you can use a combination of:

- IP Filtering to control access to your machine
- Network Address Translation (NAT)—known to Linux users as IP masquerading—to connect several computers through a common external interface

For more information, see **pf-faq** at *ftp://ftp3.usa.openbsd.org/pub/OpenBSD/doc/* in the OpenBSD documentation.

Chapter 14
Fine-Tuning Your System

This chapter describes how you can improve your system's performance.

Getting the system's status

The QNX Neutrino RTOS includes various utilities that you can use to fine-tune your system.

`hogs`

List the processes that are hogging the CPU

`pidin` (Process ID INfo)

Display system statistics

`ps`

Report process status

`top`

Display system usage (Unix)

For details about these utilities, see the *Utilities Reference*.

For more detailed and accurate data, use `tracelogger` and the System Analysis Toolkit (see the SAT *User's Guide*). The SAT logs *kernel events*, the changes to your system's state, using a specially instrumented version of the kernel (`procnto*-instr`).

If you have the Integrated Development Environment on your system, you'll find that it's the best tool for determining how you can improve your system's performance. For more information, see the IDE *User's Guide*.

Improving performance

If you run `hogs`, you'll get a rough idea of which processes are using the most CPU time.

For example:

```
$ hogs -n -% 5
PID              NAME  MSEC  PIDS  SYSTEM
1                      1315   53%    43%
6           devb-eide   593   24%    19%
54358061         make   206    8%     6%

1                      2026   83%    67%
6           devb-eide   294   12%     9%

1                      2391   75%    79%
6           devb-eide   335   10%    11%
54624301    htmlindex   249    7%     8%

1                      1004   24%    33%
54624301    htmlindex  2959   71%    98%

54624301    htmlindex  4156   96%   138%

54624301    htmlindex  4225   96%   140%

54624301    htmlindex  4162   96%   138%

1                        71   35%     2%
6           devb-eide    75   37%     2%

1                      3002   97%   100%
```

Let's look at this output. The first iteration indicates that process 1 is using 53% of the CPU. Process 1 is always the process manager, `procnto`. In this case, it's the idle thread that's using most of the CPU. The entry for `devb-eide` reflects disk I/O. The `make` utility is also using the CPU.

In the second iteration, `procnto` and `devb-eide` use most of the CPU, but the next few iterations show that `htmlindex` (a program that creates the keyword index for our online documentation) gets up to 96% of the CPU. When `htmlindex` finishes running, `procnto` and `devb-eide` use the CPU while the HTML files are written. Eventually, `procnto`—including the idle thread—gets almost all of the CPU.

You might be alarmed that `htmlindex` takes up to 96% of the CPU, but it's actually a good thing: if you're running only one program, it *should* get most of the CPU time.

If your system is running several processes at once, `hogs` could be more useful. It can tell you which of the processes is using the most CPU, and then you could adjust the priorities to favor the threads that are most important. (Remember that in QNX Neutrino, priorities are a property of threads, not of processes.) For more information, see "*Priorities*" in the Using the Command Line chapter.

Here are some other tips to help you improve your system's performance:

- You can use `pidin` to get information about the processes that are running on your system. For example, you can get the arguments used when starting the process, the state of the process's threads, and the memory that the process is using.
- The number of threads doesn't effect system reaction time as much as the number of threads at a given priority. The key to performing realtime operations properly is to set up your realtime threads with the priorities required to ensure the system response that you need.

Fine-tuning USB storage devices

If your environment hosts large (e.g., media) files on USB storage devices, you should ensure that your configuration allows sufficient RAM for read-ahead processing of large files, such as MP3 files.

You can change the configuration by adjusting the `cache` and `vnode` values that `devb-umass` passes to `io-blk.so` with the `blk` option.

A reasonable starting configuration for the `blk` option is: `cache=512k,vnode=256`. You should, however, establish benchmarks for key activities in your environment, and then adjust these values for optimal performance.

Chapter 15 Understanding System Limits

Resources on a system tend to be finite (alas!), and some are more limited than others. This chapter describes some of the limits on a QNX Neutrino system.

Let's start by considering the limits on describing limits.

QNX Neutrino is a microkernel OS, so many things that might be a core limit in some operating systems instead depend on the particular manager that implements that service under QNX Neutrino, especially for filesystems, where there are multiple possible filesystems.

Many resources depend on how much memory is available. Other limits depend on your target system. For example, the virtual address space for a process can vary by processor: 2 GB on ARM, 3.5 GB on x86 (32-bit), and 512 GB on AARCH64 and x86 (64-bit).

Some limits are a complex interaction between many things. To quote the simple/obvious limit is misleading; describing all of the interactions can be complicated. The key thing to remember while reading this chapter is that there can be many factors behind a limit.

Configurable limits

When you're trying to determine your system's limits, you can get the values of *configurable limits*, special read-only variables that store system information.

QNX Neutrino also supports *configuration strings*, which are similar to, and frequently used in conjunction with, environment variables. For more information, see the *Configuring Your Environment* chapter.

You can use the POSIX `getconf` utility to get the value of a configurable limit or a configuration string. Since `getconf` is a POSIX utility, scripts that use it instead of hard-coded QNX Neutrino-specific limits can adapt to other POSIX environments.

Some configurable limits are associated with a path; their names start with `_PC_`. When you get the value of these limits, you must provide the path (see "*Filesystem limits*," below). For example, to get the maximum length of the filename, type:

```
getconf _PC_NAME_MAX pathname
```

Other limits are associated with the entire system; their names start with `_SC_`. You don't have to provide a path when you get their values. For example, to get the maximum number of files that a process can have open, type:

```
getconf _SC_OPEN_MAX
```

In general, you can't change the value of the configurable limits—they're called "configurable" because the system can set them.

The QNX Neutrino libraries provide various functions that you can use in a program to work with configurable limits:

pathconf()

Get the value of a configurable limit that's associated with a path.

sysconf()

Get the value of a limit for the entire system.

setrlimit()

Change the value of certain limits. For example, you can use this function to limit the number of files that a process can open; this limit also depends on the value of the `-F` option to `procnto`.

Filesystem limits

Under the QNX Neutrino RTOS, filesystems aren't part of the kernel or core operating system; they're provided by separately loadable processes or libraries.

This means that:

- There's no one set limit or rule for filesystems under QNX Neutrino; the limits depend on the filesystem in question and on the process that provides access to that filesystem.
- You can provide your own filesystem process or layer that can almost transparently override or change many of the underlying values.

The sections that follow give the limits for the supported filesystems. Note the following:

- Lengths for filenames and pathnames are in bytes, not characters.
- Many of the filesystems that QNX Neutrino supports use a 32-bit format. This means that files are limited to 2 GB – 1 bytes. This, in turn, limits the size of a directory, because the file that stores the directory's information is limited to 2 GB – 1 bytes.

Querying filesystem limits

You can query the path-specific configuration limits to determine some of the properties and limits of a specific filesystem.

The full list of these limits is given in the reference for the *pathconf()* QNX Neutrino library function. The supported configurable filesystem limits include at least the following:

_PC_LINK_MAX

Maximum value of a file's link count.

_PC_MAX_CANON

Maximum number of bytes in a terminal's canonical input buffer (edit buffer).

_PC_MAX_INPUT

Maximum number of bytes in a terminal's raw input buffer.

_PC_NAME_MAX

Maximum number of bytes in a filename (not including the terminating null).

_PC_PATH_MAX

Maximum number of bytes in a pathname (not including the terminating null).

_PC_PIPE_BUF

Maximum number of bytes that can be written atomically when writing to a pipe.

_PC_CHOWN_RESTRICTED

If defined (not -1), indicates that the use of the *chown()* function is restricted to a process with appropriate privileges, and to changing the group ID of a file to the effective group ID of the process or to one of its supplementary group IDs.

_PC_NO_TRUNC

If defined (not -1), indicates that the use of pathname components longer than the value given by _PC_NAME_MAX will generate an error.

_PC_VDISABLE

If defined (not -1), this is the character value that can be used to individually disable special control characters in the `termios` control structure.

For more information, see "*Configurable limits*," above.

Power-Safe (`fs-qnx6.so`) filesystem

The limits for Power-Safe filesystems (supported by **fs-qnx6.so**) include:

Physical disk sector

32-bit (2 TB), using the `devb` API.

Logical filesystem block

512, 1024, 2048, or 4096 (set when you initially format the filesystem).

Filename length

510 bytes (UTF-8). If the filename is less than 28 bytes long, it's stored in the directory entry; if it's longer, it's stored in an external file, and the directory entry points to the name.

Pathname length

PATH_MAX (1024) bytes, not including the mountpoint or the terminating NUL.

Maximum file size

64-bit addressing.

With a 1 KB (default) block size, you can fit 256 block pointers in a block, so a file that's $16 \times 256 \times 1$ KB (4 MB) requires 1 level of indirect pointers. If the file is bigger, you need two levels (i.e., 16 blocks of 256 pointers to blocks holding another 256 pointers to blocks), which gives a maximum file size of 1 GB. For three levels of indirect pointers, the maximum file size is 256 GB.

If the block size is 2 KB, then each block holds up to 512 pointers, and everything scales accordingly.

Maximum number of files

The same as the maximum number of inodes.

This number can be set by either `mkqnx6fs` (with the `-i` option), or `mkqnx6fsimg` (with the `num_inodes` attribute in the buildfile). Both utilities assign default values if the number isn't specified.

You can query this value at any time with `df -g` *mountpoint*, where the variable is set to the Power-Safe filesystem mountpoint. The `chkqnx6fs -S` option also shows this value.

Linux Ext2 filesystem

The limits for Linux Ext2 filesystems include:

Filename length

255 bytes.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

File size

2 GB – 1.

Directory size

2 GB – 1; directories are files with inode and filename information as data.

Filesystem size

2 GB × 512.

Disk size

2^{64} bytes; limited by the disk driver.

DOS FAT12/16/32 filesystem

The limits for DOS FAT12/16/32 filesystems include:

Filename length

255 characters.

Pathname length

1024 characters, not including the mountpoint or the terminating NUL.

File size

4 GB – 1; uses a 32-bit filesystem format.

Directory size

Depends on the type of filesystem:

- The root directory of FAT12/16 is special, in that it's pregrown and can't increase. You choose the size when you format, and is typically 512 entries. FAT32 has no such limit.
- FAT directories are limited (for DOS-compatability) to containing 64 K entries.
- For long (non-8.3) names, a single filename may need multiple entries, thus reducing the possible size of a directory.

Filesystem size

Depends on the FAT format:

- for FAT12, it's 4084 clusters (largest cluster is 32 KB, hence 128 MB)
- for FAT16, it's 65524 clusters (thus 2 GB)

- for FAT32, you get access to 268435444 clusters (which is 8 TB)

Disk size

Limited by the disk driver and `io-blk`.

These filesystems don't really support permissions, but they can emulate them.

NFS2 and NFS3 filesystem

The limits for NFS2 and NFS3 filesystems include:

Filename length

255 bytes.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

File size

2 GB – 1; 32-bit filesystem limit.

Directory size, filesystem size, and disk size

Depends on the server; 32-bit filesystem limit.

CIFS filesystem

The limits for CIFS filesystems include:

Filename length

255 bytes.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

File size

2 GB – 1; 32-bit filesystem limit.

Directory size, filesystem size, and disk size

32-bit filesystem limit.

The CIFS filesystem doesn't support `chmod` or `chown`.

Embedded (flash) filesystem

The limits for embedded (flash) filesystems include:

Filename length

255 bytes.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

File size, filesystem size, and disk size

2 GB – 1.

Directory size

Limited by the available space.

Flash filesystems use a cache to remember the location of extents within files and directories, to reduce the time for random seeking (especially backward).

Embedded Transaction filesystem (ETFS)

The limits for ETFS are:

Filename length

91 bytes.

Pathname length

256 bytes, not including the mountpoint or the terminating NUL.

File size

2 GB – 1; 32-bit filesystem limit.

Absolute maximum number of files

32768 (15 bits).

Default maximum number of files

4096 (controlled by the driver's `-f` option; see the entry for **fs-etfs-ram** in the *Utilities Reference*).

Filenames that are more than 32 bytes long use two directory entries, reducing the number of files that you can actually have.

Max cluster size

4096.

Maximum filesystem size

64 GB.

For NAND flash, some additional limitations apply:

- Single-level cell (SLC) and multi-level cell (MLC) NAND flash are supported. MLC NAND requires hardware error-correction code (ECC).
- The maximum filesystem size is 4 GB.

This is a practical limit, not an actual one. When the ETFS driver starts, it scans the entire partition, recreating its own representation of the data; the larger the partition, the longer this takes.

- ECC protection of the spare area is supported only on 2 KB and 4 KB page NAND.
- The software ECC supports only 1-bit error correction, for each 256-byte buffer.
- Only NAND flash with page sizes of 512, 2048, and 4096 bytes are supported.

For ETFS on NAND, you can perform 1-bit software error correction coding (ECC) for the data in the spare area. Support configurations are available for:

- 2 KB page NAND flash devices
- 4 KB page NAND flash devices

Once calculated, the spare area receives the ECC value from *devio_postcluster()*, and then writes it to NAND flash. To determine the appropriately sized ECC value, use the following:

- For 512 NAND, it's not available
- For 2048 NAND, use 64 byte ECC
- For 4096 NAND, use 128 byte ECC

To take advantage of the spare area, you'll need to make the following changes for BSPs:

- For *devio_readtrans()* and *devio_readcluster()*—When reading the spare area, first save the spare area ECC, and then set those fields of the spare structure to 0xFF, which is required for calculating the cyclic redundancy check (CRC—data integrity checks for NAND). Perform the CRC calculation and if it fails, then in order to recover, you must attempt using the new spare area ECC value. If the spare area CRC is correct, then you can skip the ECC operation. If the ECC can correct the spare area, then set *tacode* in the transaction structure to ETFS_TRANS_ECC. If the ECC can't be corrected, then set the *tacode* to ETFS_TRANS_DATAERR.
- For *devio_postcluster()*—After calculating the CRC and ECC for the cluster data, and calculating the CRC for the spare area, add a calculation for the ECC of the spare area. When doing the CRC calculation, use 0xFF as placeholder values for the spare area ECC.

UDF filesystem

The limits for UDF filesystems include:

Filename length

255 Unicode characters.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

Disk size

This filesystem uses a 32-bit block address, but the filesystem is 64-bit (> 4 GB). We don't allow the creation of anything via `fs-udf.so`; it's read-only.

Apple Macintosh HFS and HFS Plus

The limits for the Apple Macintosh HFS (Hierarchical File System) and HFS Plus include:

Filename length

31 MacRoman characters on HFS; 255 bytes (Unicode) on HFS Plus.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

Disk size

This filesystem uses a 32-bit block address, but the filesystem is 64-bit (> 4 GB). We don't allow the creation of anything via `fs-mac.so`; it's read-only.

Windows NT filesystem

The limits for Windows NT filesystems include:

Filename length

755 characters.

Pathname length

1024 bytes, not including the mountpoint or the terminating NUL.

File size

16 TB - 64 KB; uses a 64-bit filesystem format.

Filesystem size

2^{64} - 1 clusters.

Disk size

Limited by the disk driver and `io-blk`.

This filesystem is read-only.

Other system limits

These limits apply to the entire system:

Processes

A maximum of 4094 active at any time.

Prefix space (resource-manager attaches, etc.)

Limited by memory.

Sessions and process groups

4094 (since you need at least one process per session or group).

Physical address space

No limits, except those imposed by the hardware; see the documentation for the chip you're using.

These limits apply to each process:

- Number of threads: 32767
- Number of timers: 32767
- Priorities: 0 through 255

  Priority 0 is used for the idle thread; by default, priorities of 64 and greater are privileged, so only processes with an effective user ID of 0 (i.e., **root**) can use them. Non-**root** processes can use priorities from 1 through 63.

  You can change the range of privileged priorities with the `-P` option for `procnto`. In QNX Neutrino 6.6 or later, you can append an `s` or `S` to this option if you want out-of-range priority requests by default to use the maximum allowed priority (reach a "maximum saturation point") instead of resulting in an error.

File descriptors

The total number of file descriptors has a hard limit of 32767 per process, but you're more likely to be constrained by the `-F` option to `procnto` or the RLIMIT_NOFILE system resource. The default value is 1000; the minimum is 100.

- Sockets, named semaphores, and message queues all use file descriptors. Connection IDs (coids) for side channels and server connection IDs (scoids) are returned from a different space than file descriptors, so they don't count towards the limit set by the `procnto -F` option or RLIMIT_NOFILE.
- In the traditional implementation of message queues (`mqueue`), each call to *mq_open()* uses one file descriptor; in the alternate implementation (`mq`), each call to *mq_open()* uses two.

To determine the current limit, use the `ksh` builtin command, `ulimit`, (see the *Utilities Reference*), or call *getrlimit()* (see the QNX Neutrino *C Library Reference*).

Synchronization primitives

There are no limits on the number of mutexes and condition variables (condvars).

There's no limit on the number of unnamed semaphores, but the number of named semaphores is limited by the number of available file descriptors (see "*File descriptors*," above).

TCP/IP limits

The number of open connections and sockets is limited only by memory and by the maximum number of file descriptors per process (see "*File descriptors*").

Shared memory

The number of shared memory areas is limited by the allowed virtual address space for a process, which depends on the target architecture.

See the RLIMIT_AS and RLIMIT_DATA resources for *setrlimit()* in the *C Library Reference*.

Message queues

The number of message queues is limited by the `-n` option to `mqueue` or `mq`. The default is 1024, and the maximum is 32768; you can query the current value with `sysconf( _SC_MQ_OPEN_MAX )`.

The default maximum number of entries in a queue, and the default maximum size of a queue entry depend on whether you're using the traditional (`mqueue`) or alternate (`mq`) implementation of message queues:

Attribute	Traditional	Alternate
Number of entries	1024	64
Message size	4096	256

For more information, see `mqueue` and `mq` in the *Utilities Reference*, and *mq_open()* in the QNX Neutrino *C Library Reference*.

In the traditional implementation, each call to *mq_open()* uses one file descriptor; in the alternate implementation, each call to *mq_open()* uses two.

Platform-specific limits

Limit	x86	x86_64	ARMv7	AArch64
System RAM (theoretical)	32 GB[a]	1 TB	128 GB[a]	1 TB

Limit	x86	x86_64	ARMv7	AArch64
System RAM (tested)	32 GB[a]	96 GB	4 GB[a]	8 GB
Physical addressing	36-bit with startup's `-x` option, 32-bit without	48-bit[b]	40-bit with startup's `-x` option, 32-bit without	48-bit[b]
Number of CPUs[b]	32	32	32[c]	32[c]
Virtual address space[d]	3.5 GB	512 GB	2 GB	512 GB

[a] You need to enable extended addressing by specifying the startup's `-x` option.

[b] May be limited further by the hardware.

[c] Limited to 8 if you're using a version 2 Generic Interrupt Controller (GIC).

[d] These are the absolute maximum limits for the virtual address space; you can reduce them by setting the RLIMIT_AS resource with *setrlimit()*.

Chapter 16
Technical Support

If you have any problems using QNX Neutrino, the first place to look for help is in the documentation.

However, what do you do if you need *more* help? The resources that are available to you depend on the support plan that you've bought. The community includes:

- forums
- the myQNX account center, where you can register your products so that you can download software and updates.
- Global Help Center—available at any time of day
- training
- an online knowledge base that you can search
- detailed hardware support lists
- free software
- and more

Some of these resources are free; others are available only if you've purchased a support plan. For more information about our technical support offerings, see the Services section of our website at *http://www.qnx.com*.

Appendix A Glossary

administrator

See *superuser*.

alias

A shell feature that lets you create new commands or specify your favorite options. For example, `alias my_ls='ls -F'` creates an alias called `my_ls` that the shell replaces with `ls -F`.

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.

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`.

budget

In *sporadic* scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for `mkifs` specifying the contents and other details of an *image*, or for `mkefs` specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or "cooked" mode. In this mode the character device library performs line-editing operations on each received character. Only when a line is "completely entered"—typically when a carriage return (CR) is received—will the line of data be made available to application processes. Contrast *raw mode*.

channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a *connection* (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using *ChannelCreate()*), and then receives messages from that channel (using *MsgReceive()*). Another thread that wishes to send a message to the first thread must make a connection to that channel by "attaching" to the channel (using *ConnectAttach()*) and then sending data (using *MsgSend()*).

CIFS

Common Internet File System (aka SMB)—a protocol that allows a client computer to perform transparent file access over a network to a Windows 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.

command completion

A shell feature that saves typing; type enough of the command's name to identify it uniquely, and then press **Esc** twice. If possible, the shell fills in the rest of the name.

command interpreter

A process that parses what you type on the command line; also known as a *shell*.

compound command

A command that includes a shell's reserved words, grouping constructs, and function definitions (e.g., `ls -al | less`). Contrast *simple command*.

configurable limit

A special variable that stores system information. Some (e.g., _PC_NAME_MAX) depend on the filesystem and are associated with a path; others (e.g., _SC_ARG_MAX) are independent of paths.

configuration string

A system variable that's like an environment variable, but is more dynamic. When you set an environment variable, the new value affects only the current instance of the shell and any of its children that you create after setting the variable; when you set a configuration string, its new value is immediately available to the entire system.

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 *MsgSend()* messages over them.

console

The display adapter, the screen, and the system keyboard are collectively referred to as the *physical console*. A *virtual console* emulates a physical console and lets you run more than one terminal session at a time on a machine.

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*.

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.

DNS

Domain Name Service—an Internet protocol used to convert ASCII domain names into IP addresses. In QNX Neutrino native networking, `dns` is one of *Qnet's* builtin resolvers.

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*.

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.

Changing a bit from a 0 state into a 1 state can be accomplished only 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 *EEPROM*, *flash*, and *RAM*.

EEPROM

Electrically Erasable Programmable Read-Only Memory—a memory technology that's similar to *EPROM*, but you can erase the entire device electrically instead of via ultraviolet light. 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.

extent

A contiguous sequence of blocks on a disk.

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.

FIFO

First In First Out—a scheduling policy whereby a thread is able to consume CPU at its priority level without bounds. See also *round robin* and *sporadic*.

filename completion

A shell feature that saves typing; type enough of the file's name to identify it uniquely, and then press **Esc** twice. If possible, the shell fills in the rest of the name.

filter

A program that reads from standard input and writes to standard output, such as `grep` and `sort`. You can use a pipe (`|`) to combine filters.

flash memory

A memory technology similar in characteristics to *EEPROM* memory, with the exception that erasing is performed electrically, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64 KB bytes at a time) instead of the entire device. Contrast *EPROM* and *RAM*.

FQNN

Fully Qualified Node Name—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

The process whereby a filesystem manager recovers the space occupied by deleted files and directories. Also known as space reclamation.

group

A collection of users who share similar file permissions.

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.

hard link

See *link*.

hidden file

A file whose name starts with a dot (`.`), such as **.profile**. Commands such as `ls` don't operate on hidden files unless you explicitly say to.

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 filesystem (i.e., a flash filesystem image).

inode

Information node—a storage table that holds information about files, other than the files' names. In order to support links for each file, the filename is separated from the other information that describes a file.

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 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*.

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/ROM Monitor 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*.

link

A filename; a pointer to the file's contents. Contrast *symbolic link*.

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.

metadata

Data about data; for a filesystem, metadata includes all the overhead and attributes involved in storing the user data itself, such as the name of a file, the physical blocks it uses, modification and access timestamps, and so on.

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.

mountpoint

The location in the pathname space where a resource manager has “registered” itself. For example, a CD-ROM filesystem may register a single mountpoint of **/cdrom**.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It's typically acquired (*pthread_mutex_lock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a *critical section*).

name resolution

In a QNX Neutrino network, the process by which the *Qnet* network manager converts an *FQNN* to a list of destination addresses that the transport layer knows how to get to.

name resolver

Program code that attempts to convert an *FQNN* to a destination address.

NDP

Node Discovery Protocol—proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.

network directory

A directory in the pathname space that's implemented by the *Qnet* network manager.

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.

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.

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 media to retain information across a power-cycle. For example, a hard disk is a persistent storage media, whereas a ramdisk is not, because the data is lost when power is lost.

PIC

Programmable Interrupt Controller—a hardware component that handles IRQs.

PID

Process ID. Also often *pid* (e.g., as an argument in a function call). See also *process ID*.

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.

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.

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. You typically use these devices when a program requires a terminal for standard input and output, and one doesn't exist, for example as with sockets.

Qnet

The native network manager in the QNX Neutrino RTOS.

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.

QSS

QNX Software Systems.

quoting

A method of forcing a shell's special characters to be treated as simple characters instead of being interpreted in a special way by the shell. For example, `less "my file name"` escapes the special meaning of the spaces in a filename.

RAM

Random Access Memory—a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast *flash*, *EPROM*, and *EEPROM*.

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*.

remote execution

Running commands on a machine other than your own over a network.

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*.

root

The superuser, which can do anything on your system. The superuser has what Windows calls administrator's rights.

round robin

A scheduling policy whereby a thread is given a certain period of time (the *timeslice*) 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 *FIFO* and *sporadic*.

RTOS

Realtime operating system.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast *static linking*.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as "T_{sl}". Contrast *interrupt latency*.

session

A collection of process groups established for job-control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

shell

A process that parses what you type on the command line; also known as a *command interpreter*.

shell script

A file that contains shell commands.

simple command

A command line that contains a single command, usually a program that you want to run (e.g., `less my_file`). Contrast *compound command*.

socket

A logical drive in a flash filesystem, consisting of a contiguous and homogeneous region of flash memory.

socket

In TCP/IP, a combination of an IP address and a port number that uniquely identifies a single network process.

software interrupt

Similar to a hardware interrupt (see *interrupt*), except that the source of the interrupt is software.

sporadic

A scheduling policy 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 *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 linking

The process whereby you combine your programs 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. Contrast runtime loading.

superuser

The **root** user, which can do anything on your system. The superuser has what Windows calls administrator's rights.

symbolic link

A special file that usually has a pathname as its data. Symbolic links are a flexible means of pathname indirection and are often used to provide multiple paths to a single file. Unlike hard links, symbolic links can cross filesystems and can also create links to directories.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under QNX Neutrino. A thread is a flow of execution; it exists within the context of a *process*.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an *event*, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a *round-robin* scheduled thread. This period of time is small (four times the clock period in QNX Neutrino); programs shouldn't rely on the actual value (doing so is considered bad design).

Appendix B
Examples

This appendix includes samples of some of the files described in this guide.

Buildfile for an NFS-mounting target

Here's a sample buildfile for an NFS-mounting target.

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.

```
###########################################################################
##
## QNX Neutrino RTOS on the fictitious ABC123 board
##
###########################################################################
##
## SUPPORTED DEVICES:
##
## SERIAL:  RS-232 ports UART0 and UART1
## PCI:     4 PCI slots
## NETWORK: AMD 79C973
## FLASH:   4MB Intel Strata Flash
## USB:     UHCI USB Host Controller
##
##  - For detailed instructions on the default example configuration for
##    these devices see the "CONFIGURING ON-BOARD SUPPORTED HARDWARE"
##    section below the build script section, or refer to the BSP docs.
##  - Tip: Each sub-section which relates to a particular device is marked
##         with its tag (ex. SERIAL). You can use the search features of
##         your editor to quickly find and add or remove support for
##         these devices.
##
###########################################################################
##
## NOTES:
##
###########################################################################

###########################################################################
## START OF BUILD SCRIPT
###########################################################################

[image=0x800a0000]
[virtual=armle-v7,srec] .bootstrap = {
###########################################################################
## default frequency for 4kc is 80MHz; adjust -f parameter for different
## frequencies
###########################################################################
    startup-abc123 -f 80000000 -v
    PATH=:/proc/boot procnto-32 -v
}

[+script] .script = {
    procmgr_symlink ../../proc/boot/libc.so.4 /usr/lib/ldqnx.so.2

    display_msg Welcome to the QNX Neutrino RTOS on the ABC123 board

    #######################################################################
    ## SERIAL driver
    #######################################################################
    devc-ser8250 -e -c1843200 -b38400 0x180003f8,0x80020004 \
0x180002f8,0x80020003 &
    waitfor /dev/ser1
    reopen /dev/ser1

    slogger2 &
    pipe

    #######################################################################
    ## PCI server
    #######################################################################
    display_msg Starting PCI server...

    pci-abc123 &
    waitfor /dev/pci 4

    #######################################################################
    ## FLASH driver
    #######################################################################
    # display_msg Starting flash driver...
    #
    # devf-abc123 &

    #######################################################################
    ## NETWORK driver
    ##  - substitute your IP address for 1.2.3.4
    #######################################################################
    display_msg Starting on-board ethernet with the v6 TCP/IP stack...

    io-pkt-v6-hc -decm
    if_up -p ecm0
    ifconfig ecm0 1.2.3.4

    #######################################################################
    ## REMOTE_DEBUG (gdb or Momentics)
    ##  - refer to the help documentation for the gdb, qconn and the IDE
    ##    for more information on remote debugging
    ##  - the commands shown require that NETWORK be enabled too
    #######################################################################
    # devc-pty &
    # waitfor /dev/ptyp0 4
    # qconn port=8000



    #######################################################################
    ## USB driver
    #######################################################################
    # display_msg Starting USB driver...
    #
    # io-usb-otg -duhci &
    # waitfor /dev/io-usb/io-usb 4

    #######################################################################
    ## These env variables are inherited by all the programs which follow
    #######################################################################
    SYSNAME=nto
    TERM=qansi
    PATH=:/proc/boot:/bin:/sbin:/usr/bin:/usr/sbin
    LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll

    #######################################################################
    ## NFS_REMOTE_FILESYSTEM
    ##  - This section is dependent on the NETWORK driver
    ##  - Don't forget to properly configure and run the nfsd daemon on the
    ##    remote file server.
    ##  - substitute the hostname or IP address of your NFS server for
    ##    nfs_server. The server must be exporting
    ##    "/usr/qnx630/target/qnx6/armle-v7".
    #######################################################################
    display_msg Mounting NFS filesystem...

    waitfor /dev/socket 4
    fs-nfs3 nfs_server:/usr/qnx630/target/qnx6/armle-v7 /mnt

    [+session] ksh &
}

[type=link] /bin/sh=/proc/boot/ksh
[type=link] /dev/console=/dev/ser1
[type=link] /tmp=/dev/shmem
```

```
###########################################################################
## uncomment for NFS_REMOTE_FILESYSTEM
###########################################################################
[type=link] /bin=/mnt/bin
[type=link] /sbin=/mnt/sbin
[type=link] /usr/bin=/mnt/usr/bin
[type=link] /usr/sbin=/mnt/usr/sbin
[type=link] /lib=/mnt/lib
[type=link] /usr/lib=/mnt/usr/lib
[type=link] /etc=/mnt/etc

libc.so.2
libc.so
libm.so

###########################################################################
## uncomment for NETWORK driver
###########################################################################
devnp-ecm.so
libsocket.so

###########################################################################
## uncomment for USB driver
###########################################################################
# devu-hcd-uhci.so
# libusbdi.so

[data=c]
devc-ser8250

###########################################################################
## uncomment for REMOTE_DEBUG (gdb or Momentics)
###########################################################################
# devc-pty
# qconn

###########################################################################
## uncomment for PCI server
###########################################################################
pci-abc123
pci

###########################################################################
## uncomment for FLASH driver
###########################################################################
# devf-abc123
# flashctl

###########################################################################
## uncomment for NETWORK driver
###########################################################################
io-pkt-v6-hc
ifconfig
nicinfo
netstat
ping

###########################################################################
## uncomment for USB driver
###########################################################################
# io-usb-otg
# usb

###########################################################################
## uncomment for NFS_REMOTE_FILESYSTEM
###########################################################################
fs-nfs3

###########################################################################
## general commands
###########################################################################
```

```
ls
ksh
pipe
pidin
uname
slogger2
slog2info
slay

###########################################################################
## END OF BUILD SCRIPT
###########################################################################
```

Sample buildfile

Here's a sample buildfile for a PC-based target.

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.

```
#
# The build file for QNX Neutrino booting on a PC
#
[virtual=x86,bios +compress] boot = {
    startup-x86
    PATH=/proc/boot:/bin:/usr/bin LD_LIBRARY_PATH=/proc/boot:\
/lib:/usr/lib:/lib/dll procnto-smp
}

[+script] startup-script = {
    display_msg "  "
    display_msg "QNX Neutrino inside!"
    display_msg "  "
    procmgr_symlink ../../proc/boot/libc.so.4 /usr/lib/ldqnx.so.2

    display_msg "---> Starting PCI Services"
    seedres
    pci-server
    waitfor /dev/pci

    display_msg "---> Starting Console Manager"
    devc-con -n8
    waitfor /dev/con1
    reopen /dev/con1

    display_msg "---> Starting EIDE Driver"
    devb-eide blk cache=64M,auto=partition,vnode=2000,ncache=2000,\
noatime,commit=low dos exe=all
    waitfor /dev/hd0
    waitfor /dev/hd1

    # Mount one Power-Safe filesystem as /, and another as /home.
    # Also, mount a DOS partition and the CD drive.

    mount /dev/hd0t179 /
    mount /dev/hd1t178 /home
    mount -tdos /dev/hd1t12 /fs/hd1-dos
    mount -t udf /dev/cd0 /fs/cd0

    display_msg "---> Starting /etc/system/sysinit"
    ksh -c /etc/system/sysinit
}

libc.so.2
libc.so
libcam.so
io-blk.so
cam-disk.so
fs-qnx6.so
fs-dos.so
fs-ext2.so
cam-cdrom.so
fs-udf.so

[data=c]
seedres
pci-server
devb-eide
slogger2
```

```
ksh
devc-con
mount
```

.profile

The system runs your **.profile** whenever you log in.

When you create a new user account, the user's initial **.profile** is copied from **/etc/skel/.profile** (see *Managing User Accounts*). Here's what's in that file:

```
# default .profile
if test "$(tty)" != "not a tty"; then
echo 'edit the file .profile if you want to change your environment.'
fi
```

This profile runs the `tty` utility to get the name of the terminal that's open as standard input. If there is a terminal, **.profile** simply displays a helpful hint.

You might want to set some environment variables:

EDITOR

The path to your favorite editor (the default is `vi`).

ENV

The name of the profile that `ksh` should run whenever you start a shell.

.kshrc

Here's an example of a profile that `ksh` runs if you set the *ENV* environment variable as described above for **.profile**:

```
alias rm="rm -i"
alias ll="ls -l"
export PS1='$(pwd) $ '
```

This profile does the following:

- Uses an alias to turn on interactive mode for the `rm` command. In interactive mode, `rm` asks you for confirmation before it deletes the file. The `cp` and `mv` commands also support this mode.
- Creates an alias, `ll`, that runs `ls` with the `-l` set. This gives a long listing that includes the size of the files, the permissions, and so on.
- Changes the primary prompt to include the current working directory (the default if you aren't **root** is `$`). You can also change the secondary prompt by setting *PS2*.

  Note that you should use single quotes instead of double quotes around the string. If you specify:

  ```
  export PS1="$(pwd) $ "
  ```

  the `pwd` command is evaluated right away because double quotes permit command substitution; when you change directories, the prompt doesn't change.

PPP with CHAP authentication between two QNX Neutrino boxes

The following script starts the Point-to-Point Protocol daemon, `pppd`, with a `chat` script, waits for the modem to ring, answers it, and starts PPP services with CHAP (Challenge-Handshake Authentication Protocol) authentication. After PPP services have terminated, or an error on modem answer occurs, it restarts and waits for the next call.

```
#!/bin/sh

SERIAL_PORT=$1
DEFAULT_SERIAL_PORT=/dev/ser1
PPPD="/usr/sbin/pppd"
DO_CHAT="chat -v ABORT BUSY ABORT CARRIER ABORT ERROR \
 TIMEOUT 32000000 RING ATA TIMEOUT 60 CONNECT \d\d\d"
STTY="/bin/stty"
ECHO="/bin/echo"
LOCAL_IP=10.99.99.1
REMOTE_IP=10.99.99.2

if [ "$SERIAL_PORT" == "" ]; then
    SERIAL_PORT=$DEFAULT_SERIAL_PORT
fi

#do some initialization
$STTY +sane +raw < $SERIAL_PORT

while [ true ]; do
    $ECHO "Waiting on modem $SERIAL_PORT..."
    $ECHO "Starting PPP services..."
    $PPPD connect "$DO_CHAT" debug nodetach auth +chap \
$LOCAL_IP:$REMOTE_IP $SERIAL_PORT
done;
```

The `TIMEOUT` is 32000000 because it's a long period of time before the timeout takes effect; `chat` doesn't allow an infinite wait. The **/etc/ppp/chap-secrets** is as follows:

```
# Client  Server  Secret    Addresses allowed
###############################################################
*  *  "password" *
```

You can also extend the `chat` script that answers the modem to be a little more robust with specific events that should restart the answering service other than the events given. You might want to add other features as well.

Here's the buildfile used to set up a machine to allow `telnet` connections (to log in for shell access) and `tftp` access (for file transfer) over PPP:

```
[virtual=x86,bios +compress] .bootstrap = {
    startup-x86 -K8250.2f8^0.57600.1843200.16 -v
    PATH=/proc/boot procnto-smp-instr -vvv
}
[+script] startup-script = {
    seedres
    pci-server &
    waitfor /dev/pci
    # Start 1 keyboard console
    devc-con -n8 &
    # Start serial A driver
    waitfor /dev/con1
    reopen /dev/con1
    devc-ser8250 -e -b38400
    waitfor /dev/ser1
    pipe
    touch /tmp/syslog
    syslogd
    devc-pty
    io-pkt-v4-hc -pabc100
    if_up -p abc0
    inetd &

    display_msg "[Shell]"
    [+session] PATH=/bin:/proc/boot /bin/sh &
}

# Make /tmp point to the shared memory area...
[type=link] /tmp=/dev/shmem

# Programs require the runtime linker (ldqnx.so) to be at
# a fixed location
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[type=link] /bin/sh=/bin/ksh

# We use the "c" shared lib (which also contains the
# runtime linker)
libc.so
libsocket.so

# The files above this line can be shared by multiple
# processes
[data=c]
devc-con
devc-ser8250
devc-pty
pci-server
seedres
pipe
io-pkt-v4-hc
/bin/echo=echo
/bin/stty=stty
tail
pci
chat
ifconfig
ping
syslogd
touch
./modem_ans_ppp.sh

#Services (telnetd etc) config
inetd
/usr/sbin/telnetd=telnetd
/usr/sbin/tftpd=tftpd
/usr/sbin/pppd=pppd
/bin/login=login
/bin/ksh=ksh

/etc/ppp/chap-secrets = {
# Client    Server     Secret     Addrs
########################################
*           *           "password"  *
}
/etc/syslog.conf = {
*.*     /tmp/syslog
}

# Inetd config Files
/etc/services= /etc/services
/etc/protocols= /etc/protocols
```

```
/etc/termcap= /etc/termcap
/etc/passwd= /etc/passwd
/etc/default/login= /etc/default/login
/etc/resolv.conf= /etc/resolv.conf
/etc/nsswitch.conf= /etc/nsswitch.conf
/etc/shadow = /etc/shadow

/etc/inetd.conf = {
telnet      stream  tcp nowait  root    /usr/sbin/telnetd   in.telnetd
tftp        dgram   udp wait    root    /usr/sbin/tftpd     in.tftpd
}

/etc/hosts = {
127.1   localhost.localdomain   localhost
10.99.99.1  server  server
10.99.99.2  client  client
}
```

To build the image using this buildfile, you'll need to be **root**, because it takes a copy of **/etc/passwd** and **/etc/shadow** (which make passwords easy to remember) but you can also put your own version of them into the buildfile as inline files.

Using two computers with modems, you can have one automatically answer, establish PPP services, and authenticate. You can then `telnet` and `tftp` to the server from a client. Use these client `pppd` parameters (in addition to the same **chap-secrets** file):

```
pppd connect "chat -v -f/tmp/dial_modem" auth +chap /dev/ser3
```

but use the appropriate serial port for the client-side modem instead of **/dev/ser3**. Make sure you use the full path to your modem script. The `chat` script, **dial_modem**, is fairly simple:

```
ABORT 'NO CARRIER'
ABORT 'ERROR'
ABORT 'BUSY'

'' ATDTxxxxxxx
CONNECT ''
```

Index