QNX® Realtime Operating System

Installation and Configuration

For QNX® 4.25

Printed under license by:

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

Publishing history	
June 1996	First edition
December 1997	Second edition

Electronic edition published 2005.

Technical support options

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

Contents

About This Guide

The *Installation & Configuration* guide accompanies the QNX operating system and is intended for both system administrators and end-users. It contains the information you'll need to install, configure, and maintain a QNX system.

The following table may help you find information quickly:

When you want to:	**Go to this chapter:**
Check software installation prerequisites	Basic Installation
Install the software onto a hard disk	″
Influence what happens when QNX boots	″
Set up the initial working environment	″
Learn about text-mode keyboard conventions	QNX Console & Keyboard Conventions
Use virtual consoles	″
Locate system files	Where QNX Files Are Found
Learn about QNX modules and system services	Building an OS Image
Assign initial priorities to system processes	″
Select and change the services built into the OS	″
Build a custom OS image	″

continued…

When you want to:	Go to this chapter:
Mount and unmount partitions as filesystems	Setting up Filesystems
Delete block special devices and reinstall filesystem drivers	″
Use prefixes or symbolic links	″
Set up a DOS filesystem	″
Start serial and parallel device drivers	Connecting Character Devices
Assign a driver to a parallel or serial port	″
Configure hardware I/O addresses and interrupts	″
Configure a modem to answer incoming calls	″
Troubleshoot a serial connection problem	″
Get information about installed licenses	Licensing
Add and activate new licenses	″
Transfer a license from one node to another	″
Plan your network	Networking
Install network cards and drivers	″

continued...

When you want to:	Go to this chapter:
Assign logical node and network IDs	″
Configure a boot server	″
Configure a machine equipped with a boot ROM	″
Configure a machine to boot from its own hard disk	″
Run network diagnostics	″
Add new users to the system	Setting up User Accounts
Set up working environment for users	″
Control access to system resources	″
Maintain user accounts	″
Change information in the password database	″
View and change user permissions	″
Set up system accounting information and event logging	″
Select a terminal type	Setting up Terminals
Change terminal capabilities	″

continued...

When you want to:	Go to this chapter:
Create a custom **`terminfo`** file	″
Share printer resources	Print Spooling
Set up and manage spooling services	″
Set up a data-enqueuing facility	″
Configure queues, filters, and targets	″
Submit, query, suspend, or cancel spool jobs	″
Plan your backup strategy	Making Backups
Select suitable backup media and archive formats	″
Compress and archive files	″
Retrieve a set of archived files	″
Maintain files	Disk & File Recovery
Check data integrity	″
Recover data	″
Fix a system that won't boot	″
Register your service plan and software	″

continued...

When you want to:	Go to this chapter:
Update your software	″
Download free software	″
Get technical support	″
Find out about training	″

What's new online (as of August 05, 2005)

"Tape" section in Making Backups chapter:

Added note about the filesystem's 512-byte record limit. Also, gave example of increasing the blocking factor when using the **pax** command and tape devices.

Chapter 1

Basic Installation

In this chapter. . .

Installing QNX onto a hard disk

Before you proceed with the installation, be sure to read the installation instructions as well as the *Release Notes* that came with your software. The *Release Notes* contain important information that could influence the way you choose to install your software.

You should also read the *System Architecture* manual, which explains the basic philosophy and operation of QNX. We assume you have this basic level of familiarity with the OS.

System requirements

System requirements vary, depending on which products you wish to install. For example, to install the base OS on your development system, you'll need at least 14M free disk space. For more information, see the installation instructions that are shipped with each product.

Your target system will likely require far less disk space (if it even has a disk!), RAM, etc. than your development system. Since QNX is a modular OS, you can easily configure your target system to use only the modules actually required at run time.

Distribution media

QNX products may be shipped on either of these two media:

- diskette
- CD-ROM

In both cases, you'll use a QNX installation program to install QNX onto your hard disk. The program creates a directory structure, copies the software to your hard disk, and builds an OS boot image configured according to your input during the installation process.

During the installation procedure, you may be prompted to confirm your hardware configuration. Make sure you have the correct information about your hardware before you begin:

- hard disk size and partition(s)
- hard disk controller type (IDE, Adaptec 2940 SCSI, etc.)
- network card type and manufacturer (e.g. Ethernet Novell NE2000)

CD-ROM distribution

If your QNX software was shipped on a CD, the installation procedure is quite simple:

1. Insert the QNX boot diskette into the floppy drive.
2. Insert the CD into the CD-ROM drive.
3. Reboot your computer.

When your computer boots, simply follow the instructions on your screen.

Floppy diskette distribution

To install QNX from floppies, follow these steps:

1. Place the QNX boot diskette into the floppy drive and reboot the computer.

   You should see a spinning arrow or a series of dots in the top left corner of your screen, followed by the QNX logo, a welcome message, and a shell prompt.

If you'd like information about the **`install`** options before you continue, type **`use install`** at the shell prompt.

2. To transfer the software from the diskettes to your hard disk, type the following command at the shell prompt:

   `install`

   Simply follow the instructions on your screen to set up your hard disk so it will boot QNX.

3. Reboot from hard disk.

   Once all the files have been installed from the floppy diskettes, you should remove any diskettes and reboot your computer from hard disk. QNX should now be up and running. At this point, you'll need to log in as **`root`**.

4. You're now ready to install additional software, customize your installation, set up your network, etc.

Installing additional software

There are three possible ways to install additional QSSL software products:

- from CD-ROM
- from floppies

If from CD-ROM. . .

If you have a QNX Products CD-ROM, you can install whatever products are on the CD-ROM, provided you have the appropriate licenses.

In this procedure, you'll need to enter the license information that was included in the package you purchased.

You must be running Photon to do a CD-ROM install.

1. Insert the CD-ROM.
2. Run the Package Installer.
3. From the list of available products, click on the product you want to install.
4. When prompted to do so, enter the appropriate license information for each product you're installing.
5. Follow the instructions on your screen.

If from floppies...

All additional software products supplied by QSSL for QNX come in a compressed format. You can install these products with the **/etc/install** utility shipped with the OS — just follow the instructions below.

You might also be able to use this method to install packages from third-party vendors, but always check the instructions that come with each package before you proceed.

Before installing your software, you must be logged in as the superuser (**root**), and the floppy driver must be running. To check if the driver is running, type:

```
sin ver
```

If "**Floppy**" doesn't appear in the **NAME** column, type:

```
Fsys.floppy &
```

Installation procedure

To install your additional software, follow these steps:

1. Insert the product's disk 1 in the floppy drive.
2. At the shell prompt, type:

   ```
   cd /
   /etc/install [drive]
   ```

   where *drive* is the device from which the software is being installed. The default is the local floppy drive (**/dev/fd0**).

Simply follow the instructions on your screen; the software will be installed onto your hard disk.

The system initialization file

What happens when the system boots

When QNX boots, an image composed of several QNX processes is loaded into main memory. The first process is **boot**, which does real-mode initialization and then puts the machine into protected mode.

The second process in the image is the Process Manager (**Proc32**), which contains the Microkernel. The Process Manager performs processor initialization, and then schedules each process included within the image for execution.

The last process in the image is the **sinit** utility. The **sinit** utility initiates the second phase of system initialization by starting a shell that executes commands from a file. This file — the system initialization file (**sysinit**) — contains commands that set up services for a machine. It's a standard shell script that runs just like any other shell script except that breaks are disabled.

Being able to start system services after boot is one of the benefits of QNX's modular architecture. The booted image typically contains only the few essential services needed to start all other required services.

When **sinit** runs, it firsts determines if the image it's part of was booted from disk or from over the network. If the image was booted from disk, **sinit** checks to see if a normal boot or an alternate boot occurred. For details, see the chapter on building an OS image.

You can optionally select an alternate boot by pressing Esc when prompted by the boot loader.

During a normal boot, **sinit** tries to boot from the **/etc/config/sysinit.***node* file. During an alternate boot (from local disk only), **sinit** tries to boot from the **/etc/config/altsysinit** file. If it can't boot from either of those files, **sinit** will try to boot from the **/etc/config/sysinit** file.

If **sinit** can't find or open a system initialization file, it terminates without initializing the system. If the open succeeds, **sinit** replaces itself with the shell (**/bin/sh**) and passes to it the name of the file that **sinit** opened.

Customizing the system initialization file

Each system initialization file is simply an ASCII file that you can edit using a text editor (e.g. **vi**). Each of the three types serves a particular purpose:

- **sysinit.***node* — node-specific file (where *node* is a value between 1 and the number of nodes in your network)
- **sysinit** — default file if **sinit** can't find a **sysinit.***node* file
- **altsysinit** — file that's run when booting an alternate OS from local disk

sysinit.*node*

If you wish to customize your installation, this is the file you modify. It contains the custom commands needed to set up the environment and services for a particular machine. Every node in the network can have its own setup.

A **sysinit.***node* file is always created by the **install** program. The file's initial contents reflect the installation parameters determined by the choices you made when you installed the software through **install**.

Customizing a node's setup

To customize a machine's setup, you'll need to have a **sysinit.***node* file on the boot server that boots the node you're customizing.

You can start with the **sysinit.***node* file for another node with a similar configuration, or you can simply make a copy of the default **sysinit** file, which you may then modify:

```
cp /etc/config/sysinit /etc/config/sysinit.node
```

Remember that the *node* suffix must be the logical node ID of the machine you're customizing.

If you change a machine's logical node ID, the machine will look for the **sysinit.***node* file that matches its new logical node ID.

sysinit

The **sysinit** file is executed when a **sysinit.***node* file isn't present.

This file is automatically put on your system during installation. We ship it as a generic file that should be able to boot *any* machine — we recommend that you make few or no modifications to this **sysinit** file.

altsysinit

This file serves as a safety net in case a modification to your **sysinit.***node* file leaves the system in a state where you can't log in.

The **altsysinit** file is executed only if you specify an alternate boot when booting from local disk (i.e. you press Esc when the boot loader prompts you for an alternate boot).

The **altsysinit** file should always contain a backup of the last working copy of the **sysinit.***node* file for a machine that boots from a local hard disk. So, before you make any changes to a working **sysinit.***node* file that could prevent the machine from booting from hard disk, you should copy the **/.boot** file to **/.altboot** and copy the **sysinit.***node* file to **altsysinit**:

```
cp /.boot /.altboot
cp /etc/config/sysinit.node /etc/config/altsysinit
```

We recommend that you update the **/.altboot** and **altsysinit** files after all successful changes to your **sysinit.***node* file.

Using the system initialization file

Base-level services

The contents of each machine's **sysinit** file reflect the hardware on that machine and the services it will provide. The following describes a base set of services that will be in most **sysinit** files.

If several machines will be using a common set of commands, you can place the commands in a separate shell script and have the file executed via the **.** (dot) shell built-in. For example, let's say you create a file called **techies** containing commands to be used by all the machines in your Technical Department. The file could be executed from within the **sysinit.***node* file of each machine in that department with the following command:

```
. /etc/config/techies
```

You'd add node-specific commands after this "dot" line. For more information on the dot command, see **sh** in the *Utilities Reference*.

Establishing the time zone

The following lines define the time zone (EST in this case) and get the time from the realtime clock. These two lines should be the first in the file for machines that boot from hard disk. Note that the **rtc** line is optional for a machine that boots over the network.

```
export TZ=EST5EDT
rtc hw
```

For more information, see "Time zones and the realtime clock" at the end of this chapter.

Starting device drivers

The following lines start the Device Manager (**Dev**) and the console driver (**Dev.ansi**) with eight virtual consoles, then instruct the shell to reopen its standard I/O through the new console device:

```
Dev &
Dev.ansi -n 8 &
reopen //0/dev/con1
```

The following lines start the serial driver **Dev.ser**, which looks for COM1 and COM2, and the parallel driver **Dev.par**, which looks for LPT1 and LPT2. These drivers terminate if they can't find the necessary hardware.

```
Dev.ser &
Dev.par &
```

If you started **Dev.ser**, you might need to use the **stty** utility to change the serial port configuration. For example, the following line changes the baud rate to 57600:

```
stty baud=57600 </dev/ser1
```

Starting the floating-point emulator

If your programs use floating point and you don't have a floating-point unit (FPU), you'll need to start the floating-point emulator:

```
emu87 &
```

Loading node mappings

When booting a node on a network, you must run the **netmap** utility, which informs the Network Manager (**Net**) of node ID mappings. You should place the following **netmap** command in the **sysinit.***node* file, even if the node isn't currently running on a network (the command has no effect on a non-networked machine):

```
netmap -f
```

Note that this command is included in the standard system initialization files shipped by QSSL.

Starting a name server

Every machine in the network must have access to the global name server. You could start the name server on this machine if it isn't rebooted frequently:

```
nameloc &
```

The **nameloc** utility periodically communicates with each node in turn, starting with node 1 and continuing up to node *n*, where *n* is the number of installed network licenses. It's a good idea to run **nameloc** on two nodes in order to provide redundancy in case one node running **nameloc** fails.

We don't recommend running **nameloc** on every node on the network, as this causes unnecessary network traffic. For more information, see the **nameloc** utility in the *Utilities Reference*.

Starting a terminal daemon

The following line starts a login on the first console and arms all other consoles:

```
tinit -T /dev/con* -t /dev/con1 &
```

While it isn't mandatory, almost all installations use the **tinit** utility. The **sysinit** files we provide contain **tinit**.

Optional services

You can add many other services to your **sysinit** file. You should add these services just before the line containing the **tinit** command. The examples below show some commonly used optional services. Note that these utilities typically support command-line options to modify their behavior — these options are explained in the *Utilities Reference* for each utility.

Setting environment variables

Processes started in the **sysinit** file inherit their environment variables. The general syntax of an environment variable definition is as follows:

export *var*=*value*

where *var* is the name of the environment variable (such as **TERM** for terminal type or **TZ** for time zone), and *value* is the setting. The **export** command is described further under the **sh** man page in the *Utilities Reference*.

Starting a floppy driver

To start a local floppy driver, first start the Filesystem Manager **Fsys** and then the driver itself as shown below. If the QNX filesystem is already running locally (as will be the case if the machine boots from disk), the Filesystem Manager will already be running and you won't have to include the first line:

```
Fsys &
Fsys.floppy &
```

If you want to access the floppy as a QNX filesystem, you must mount it as such:

```
mount /dev/fd0 /fd0
```

Starting a CD-ROM filesystem

To access files on a CD, you'll need to run the **Iso9660fsys** filesystem manager, which supports the ISO standard for CD-ROM filesystems as well as the Rock Ridge extensions:

```
Iso9660fsys &
```

Starting a DOS filesystem

If you need to access DOS floppies and partitions, you'll have to start a DOS filesystem (**Dosfsys**):

```
Dosfsys &
```

Starting TCP/IP

If you want to connect to a non-QNX machine (e.g. for Internet access), you'll need to start the TCP/IP socket manager. For details, refer to the documentation for the **TCP/IP for QNX** package.

Starting a cron server

If the machine will rarely be rebooted, consider making it a **cron** server:

```
cron &
```

Starting a mouse driver (for console-based apps)

Some text-based applications (e.g. **vedit**) let you use a mouse. When using these apps on a text-mode console (i.e. without Photon), you'll need to run the **Mouse** manager. Several mouse types are supported. The following command will detect and start the correct mouse driver:

```
mousetrap start
```

Starting the Photon microGUI

On some systems, the system initialization file launches the Photon windowing system for graphical applications. For more information, see the chapter on Photon in the QNX *System Architecture* manual.

International keyboard support

By default, the QNX console driver assumes a 101-key US keyboard layout. However, QNX provides a variety of configurations corresponding to the keyboards most commonly used throughout the world. If you need to select an alternate keyboard configuration, use the **kbd** utility.

If you select a keyboard other than the US type when installing QNX on a machine that will boot from its own hard disk, the appropriate **kbd** command will be added to the machine's **sysinit.***node* file. If you subsequently set up other nodes, you'll have to change their **sysinit.***node* files to include this command.

Note that you can also generate custom keyboard layouts with the **kedit** utility.

Time zones and the realtime clock

It's important that the correct date, time, and time zone information be established early during initialization. These values should be set in the first line of your **sysinit** file. The **install** program assumes that your hardware clock has a valid date and time and asks you for the time zone information.

Valid dates on a QNX system range from January 1970 to January 2038. The internal date and time representation reaches its maximum value in 2038. If your system must operate past 2038 and there's no way for the system to be upgraded or modified in the field, you'll have to take special care with system dates (contact us for help with this).

Some computer BIOSs, and certain earlier versions of the QNX **rtc** utility, have difficulty handling the transition between the year 1999 and the year 2000. Note also that 2-digit forms of years as input to programs may stop working in older programs when years 2000 or greater are represented this way.

Internally, QNX uses Coordinated Universal Time (UTC), sometimes called Greenwich Mean Time. Applications and utilities convert to

local time by using information from the time zone environment variable.

If the time zone environment variable **TZ** isn't set, QNX assumes that your local time is the same as North American Eastern Standard Time with current Daylight Saving Time (EST5EDT). If you wish to transfer files to another system in a different time zone, the dates on the transferred file will appear to be shifted by the difference between the two time zones.

Establishing the time zone

The time zone information should be established before the current date and time is set. If the realtime clock in your computer has been set to local time, QNX needs the time zone information in order to derive UTC.

In the following example, the time zone and the time change rules are set for Eastern Standard Time in North America:

```
export TZ=EST5EDT4,M4.1.0/3,M10.5.0/3
```

where:

`export`	shell command to set an environment variable
`TZ`	name of variable
`EST`	Eastern Standard Time
`5`	UTC - 5
`EDT`	daylight saving time
`4`	UTC - 4
`M4.1.0/3`	first Sunday of April at 3am
`M10.5.0/3`	last Sunday of October at 3am

Getting the date and time from the realtime clock

If you're booting from disk, you should follow the time zone rules in your **sysinit** file with the **rtc** utility to establish the current date and time from the realtime clock. The following two lines would accomplish this:

```
export TZ=EST5EDT4,M4.1.0/3,M10.5.0/3
rtc hw
```

Note that there are two possible approaches to take when setting the realtime clock in your machine:

- setting the realtime clock to UTC
- setting the realtime clock to local time

We recommend that you set the time in the realtime clock to UTC. But if you're also running operating systems that assume the realtime clock is set to local time (e.g. DOS), you'll want to use the **rtc** utility with the **-l** ("el") option:

```
rtc -l hw
```

This invocation of **rtc** assumes that the realtime clock was set to local time. Note that when you use local time in the realtime clock, you'll have to manually change the value in the realtime clock when you switch to and from daylight saving time in locales where it's used.

If the time in your hardware clock is incorrect (perhaps the battery has been replaced), you should set the system time using the **date** utility, then set the realtime clock using the **rtc** utility with the **-s** option. For details, see the documentation for these utilities in the *Utilities Reference*.

If you're booting over the network...

If you boot over the network, the booting machine will inherit the UTC time from its boot server. Therefore, you don't need to put this information in your **`sysinit`** file.

Chapter 2

Building an OS Image

In this chapter. . .

Introduction

QNX is a modular operating system composed of a microkernel and one or more processes that provide services. For example, the process named **Fsys** provides filesystem services, the process named **Net** provides network services, and so on.

When you build an OS image, you select the services that you want to be available immediately after boot, and include the processes that provide those services into a custom-built OS. You create this image with the **buildqnx** utility. The image can be booted from disk by the QNX partition loader or booted over the network using the **netboot** utility.

Constructing a build file

The **buildqnx** utility produces a binary image file containing several processes as listed in an input text file called a "build" file. The build files are kept in the **/boot/build** directory; the image files are kept in the **/boot/images** directory.

You can create an image for any node by invoking **buildqnx** directly or by using the **make** utility and the **Makefile** in the **/boot** directory. For example, you could enter either of the following commands to create an image file called **hard.1** from the sample build file **hard.1**:

```
cd /boot
buildqnx build/hard.1 images/hard.ata.1
```

OR

```
cd /boot
make b=hard.ata.1
```

Build file format

Each program you want included in the created image is specified using three lines in the build file:

- first line — the pathname of the program you want included
- second line — starts with a **$**, followed by an initial heap size value of 1, followed by a command.
- third line — *must* be blank to separate one entry from the next.

Relative pathnames are assumed to start at **/boot**.

Here's the sample build file **hard.ata.1**:

```
sys/boot
$ 1 boot -v

sys/Proc32
$ 1 Proc32 -l 1

sys/Slib32
$ 1 Slib32

sys/Slib16
$ 1 Slib16

/bin/Fsys
$ 1 Fsys

/bin/Fsys.ata
$ 1 Fsys.ata

/bin/mount
$ 1 mount -p /dev/hd0 /dev/hd0t77 /

/bin/sinit
$ 1 sinit TERM=ansi
```

Setting heap sizes

The number after the **$** symbol sets the initial heap size. A value of "1" will cause the heap size to grow dynamically as needed.

Setting priorities

You can set the initial priority of a process by following the heap size value with an optional **:***nn* argument, where *nn* is the desired priority in the range from 1 (the lowest) to 30 (the highest). For example, the following would start the floppy driver (**Fsys.floppy**) with a priority of 8:

```
/bin/Fsys.floppy
$ 1:8 Fsys.floppy
```

Where executables are stored

Most QNX system executables are stored in the **/bin** directory. Some components are usable only as part of the boot image; these are stored in subdirectories under the **/boot** directories.

The following table summarizes these conventions:

To find:	Look in:
system executables	**/bin**
OS image **Makefile**	**/boot**
build files to make images (the **make** utility reads these)	**/boot/build**
OS image files	**/boot/images**
system processes needed at boot time	**/boot/sys**

For more information on where files are stored in QNX, see Appendix B.

Selecting executables for an image

The processes you include in an OS image are determined by several factors. You can typically group images into three classifications:

- images loaded from disk
- images loaded over the network
- images loaded into embedded systems

The **boot** process (**/boot/sys/boot**) is always automatically inserted into the first line of the generated image, even if it isn't explicitly specified. If you need to supply options to **sys/boot**, you must add the line "**sys/boot**" to the build file as the first process.

For images loaded from disk or over the network, you can start most processes after booting. You do so by placing their command lines in the system initialization file that QNX executes after boot (see "The system initialization file" in the Basic Installation chapter). This lets you keep the boot image small (**<** 512K) and simple.

The build file doesn't generally contain the Device Manager **/bin/Dev**. The Device Manager and its drivers are usually started in the system initialization file *after* the system boots.

If the system initialization file isn't executed, the Device Manager will *not* be started. As a result, your keyboard and system console won't function.

Mandatory processes

When you're building an image, remember that there are two mandatory processes:

- the Process Manager/Microkernel (**/boot/sys/Proc32**)
- the system shared library (**/boot/sys/Slib32**)

To run 16-bit programs, you must include the 16-bit shared library **/boot/sys/Slib16**.

Disk images

For hard disk booting, you need to include:

- the two required system processes (**Proc32** first and **Slib32** second)
- the Filesystem Manager (**Fsys**)
- the driver required to access the drive (**Fsys.***driver*)

For more information, see **buildqnx**, **Fsys***, **Proc32**, **Slib***, and **sinit** in the *Utilities Reference*.

The Makefile for disk booting

The **Makefile** for building boot images (under the directory **/boot**) contains an entry for making a generic hard disk boot image. When you build the OS image, default parameter values are used unless you enter new values when you invoke the **make** utility.

Here's a **make** example that will create a boot image called **/boot/images/hard.1** using a build file called **/boot/build/hard.1**:

```
cd /boot
make b=hard.1
```

Note that you can create images statically by invoking the **buildqnx** utility or dynamically by running the **netboot** utility. For details, see these utilities in the *Utilities Reference*.

Here's an example of invoking **buildqnx** "by hand" — this will create an image called **/boot/images/ws32.ether1000** from a build file called **/boot/build/ws32.ether1000**:

```
cd /boot
buildqnx build/ws images/ws32.ether1000
```

Copying an image to `/.boot`

Once you've built an image, it won't become the new boot image until you copy it to the **`/.boot`** file. However, before you do this, you should make a backup of the current **`/.boot`** file by copying it to the **`/.altboot`** file:

```
cp /.boot /.altboot
```

If for any reason your new image doesn't work properly, you can press Esc when prompted during the boot process and load the **`/.altboot`** file instead of the **`/.boot`** file.

When you select the alternate boot image, the normal check for the **`/etc/config/sysinit.`***node* file is replaced by a check for the **`/etc/config/altsysinit`** file. You should ensure that the **`altsysinit`** file contains the latest copy of your working **`sysinit`** file:

```
cp /etc/config/sysinit.node /etc/config/altsysinit
```

Network images

For network booting, you need to include:

- the two required system processes (**`Proc32`** first and **`Slib32`** second)
- the Network Manager (**`Net`**)
- the driver required to access the network hardware (**`Net.`***driver*)

The build file for network booting

The following build file (**`/boot/build/ws.ether1000`**) contains parameter markers that inherit their values from the network environment:

```
sys/Proc32
$ 1 Proc32 -l $(lnode) -D

sys/Slib32
$ 1 Slib32

sys/Slib16
$ 1 Slib16

/bin/Net
$ 1 Net -m $(netmap)

/bin/Net.ether1000
$ 1 Net.ether1000

/bin/sinit
$ 1 sinit -r //$(bnode)/  TERM=qnx TZ=$(TZ)

sys/Debugger32
$ 1 Debug
```

The **$(lnode)** marker in the **Proc32** command takes as its value the logical node ID of a booting machine. The logical node ID is determined by **netboot** and passed to **buildqnx** based on the Ethernet address of the booting machine and its mapping entry in the **/etc/config/netmap** file.

The **$(netmap)** marker in the **Net** command tells **Net** the initial network mapping of the boot server so that the booting machine can start network communications with it.

The **$(bnode)** marker in the **sinit** command represents the logical node ID of the boot server. The command sets the root (**/**) of the booting machine's filesystem to the root of the boot server.

The value of the **$(TZ)** marker is inherited from the boot server. The command sets the booting machine's time zone information to match the current setting of the boot server's **TZ** environment variable.

If you must build a pre-built image for a node, don't use markers — hardcode the required values directly into a custom build file instead. For more information, see **buildqnx** in the *Utilities Reference* as well as the Networking chapter in this guide.

When you boot over the network, you have the option of loading a pre-built image or building one on the fly. If you build an image on the fly — which is recommended — then you won't need to build one manually as shown above. This option is specified in the **/etc/config/netboot** file and documented in the **netboot** utility man page.

When the **netboot** utility invokes **buildqnx** to build an image on the fly, the image file is *not* written to disk.

Embedded images

An embedded image is required for embedded CPU systems. While the term "embedded system" has various meanings, QNX supports two general classifications:

- embedded PCs
- custom hardware

Embedded PCs

These are computerized controllers that behave just like a PC — they can have keyboards, screens, disks, or network cards connected to them. They're typically used in process control applications. Some of these systems have a small amount of Flash memory available, either built into the CPU card (such as the Ziatech ZT8902) or available as a plug-in card (such as the Ampro MM/SSD).

The QNX Embedded Kit supports various CPU cards and Flash cards. This support includes booting from the Flash card (usually via some BIOS hook) and using the Flash memory as a filesystem.

Custom hardware

These systems are custom designed to fit one specific application. They usually have Flash memory, but no disk or network card. Most of these kinds of systems (e.g. Intel 386EX and AMD SC400 processors) have no keyboard or screen. The QNX Embedded Kit provides support for some of the commonly used processor evaluation boards.

Selecting processes for an embedded image

You can modify the QNX Embedded Kit to support your particular hardware. Your embedded image should contain *only* the processes required to run the system. This image would then have to be transferred to the embedded system (most likely in ROM).

Chapter 3

Setting up Filesystems

In this chapter. . .

Introduction

Each filesystem has its own *root directory*. This root is at the top of the directory hierarchy and is where QNX starts looking for other directories and files. The typical name for the root directory of the default filesystem on a hard disk is slash (**/**).

A filesystem rooted at **/** may be composed of one or more *physical filesystems* grafted together. A physical filesystem can be a separate disk or partition.

If a hard disk has more than one filesystem, their names might be **/hd2**, **/hd3**, **/home2**, **/home3**, etc.

One of the physical filesystems is typically assigned to be the root (**/**), while the other filesystems are mounted as subdirectories.

A prefix tree maps pathnames to I/O managers and determines which disks and devices are accessed when a process opens a file. In all the examples in this chapter, the prefix tree is used to partition the namespace. For more information about the prefix tree, see the I/O Namespace chapter in *System Architecture*.

Partitioning the pathname space

The **mount** utility can be used to mount disk partitions as block special files, and to mount block special files as QNX filesystems. When you mount a block special file as a filesystem, its location in the pathname space is called a *mount point*.

For example, given a QNX partition (type 77) on hard disk 0 (**/dev/hd0**), the following command would create the block special file **/dev/hd0t77**:

```
mount -p /dev/hd0
```

The following command would mount all partitions found on hard disk 0 and mount the QNX partition as the root:

```
mount -p /dev/hd0 /dev/hd0t77 /
```

The superuser, the owner, or anyone who has write permission to a device (e.g. disk) or partition can mount or unmount that device. For example, it's a good idea to unmount floppy devices or other removable media before removing the disk from the drive. This will help prevent unusual errors or possible file corruption. For more information, see the description of **umount** in the *Utilities Reference*.

The examples that follow should help clarify how the pathname space is partitioned. We'll look at these configurations:

- a hard disk and a floppy
- two hard disks
- multiple QNX partitions
- local and remote hard disks

In these examples, we assume that **Fsys** as well as the appropriate drivers are running, and that the **mount -p** command has been done on your hard disk.

A hard disk and a floppy

The hard disk is mounted as slash (**/**) and forms the root of the filesystem. The floppy is mounted as **/fd0**:

```
mount /dev/hd0t77 /
mount /dev/fd0 /fd0
```

Any reference to a pathname starting with **/fd0** will be directed to a QNX filesystem on the floppy. For example, to show all files on the floppy:

```
ls -aR /fd0
```

Two hard disks (same node)

The first hard disk is mounted as slash (**/**) and forms the root of the filesystem. The second hard disk is mounted as **/home2**:

```
mount /dev/hd0t77 /
mount /dev/hd1t77 /home2
```

Any reference to a pathname starting with **/home2** will be directed to a QNX filesystem on the second hard drive. For example, to show all files on the second hard drive:

```
ls -aR /home2
```

Multiple QNX partitions

You can have up to *four* QNX partitions on a single hard drive. The primary partition should be type 77 (see the **fdisk** utility in the *Utilities Reference*). The rest should be assigned 78, 79, and 80. For example:

```
mount /dev/hd0t77 /
mount /dev/hd0t78 /home2
```

Any reference to a pathname starting with **/home2** will be directed to a QNX filesystem on the second partition. For example, to show all files on the second partition:

```
ls -aR /home2
```

Local and remote hard disks

In a network, you might have disks with QNX filesystems on more than one machine. You may configure the namespace for these filesystems to be:

- independent

- primary/secondary
- linked independent

In the following examples, you'll notice that local filesystems are associated with block special files (e.g. **/dev/hd0t77**), while remote filesystems are associated with pathname prefix mappings. This prefix mapping redirects requests to a remote filesystem that will be associated with a remote block special file.

Independent

In this configuration, you treat each machine as an independent, self-contained filesystem. To access a file on a remote machine, you'd precede a pathname with the remote machine's node number. For example:

```
//10/etc/motd
```

The ability to specify a filesystem for a particular node will always work and is the most general mechanism for accessing remote files.

Primary/secondary

In this configuration, you treat one filesystem as the primary and you mount the second filesystem as a subdirectory under the primary. For example, assume node 1 has the primary and node 2 has the secondary mounted as **/home2**. Node 1 boots from hard disk and has its root set to its local disk by a **mount** utility built into the OS image (with **Fsys** and a driver). Its system initialization file would invoke the **prefix** utility to mount the remote filesystem as follows:

```
prefix -A /home2=//2/home2
```

Node 2 boots over the network from node 1 and has its filesystem root set to **/=//1/** using the **-r** option of the **sinit** utility built into the OS. Its system initialization file would invoke the **mount** utility to mount the local filesystem as follows:

```
mount /dev/hd0t77 /home2
```

The Filesystem Manager (**Fsys**) and its driver may be built into the image, but they're more likely started from the system initialization file. In other words, node 2 boots like a simple diskless machine, then starts its filesystem after booting.

Both nodes 1 and 2 will access the filesystem on node 1 as **/** and the filesystem on node 2 as **/home2**.

Linked independent

In this configuration, you treat each machine as an independent self-contained filesystem, but you link portions of them together via the **prefix** utility. For example, assume that the filesystem on node 1 has a **/home1** directory while the filesystem on node 2 has a **/home2** directory. You could map each home directory into the other filesystem's space as follows:

On node 1:

```
prefix -A /home2=//2/home2
```

On node 2:

```
prefix -A /home1=//1/home1
```

Other than this link, each filesystem is self-contained with its own copies of **/bin** etc. The advantage is greater redundancy: if one department uses node 2, and node 1 goes down, the department using node 2 can continue to work (except that the files under **/home1** won't be available).

Alternatively, you could use symbolic links to access files in another filesystem. A symbolic link creates a special file that has a pathname as its data. To create a symbolic link, specify the **-s** option to the **ln** utility.

Let's say there's a file called **file1** in the **/home1** directory on node 1. The following command would let the department using node 2 access **file1** through the **file2** symbolic link in the **/home2** directory:

```
ln -s /home1/file1 //2/home2/file2
```

This creates the symbolic link **//2/home2/file2**, which points to **/home1/file1**. You could display the contents of **file1** within the **/home2** directory on node 2 using the **less** utility as follows:

```
less file2
```

For more information about symbolic links, see the Filesystem Manager chapter in *System Architecture*.

Removing and reinstalling drivers

The superuser can delete (typically via the **rm** utility) block special devices. If all the devices that a driver is responsible for are deleted, the driver will terminate automatically. This allows users with the appropriate privileges to install, remove, and reinstall drivers in a running system.

For information about how to change user permissions, see the "File permissions" section in the chapter on Setting up User Accounts.

Setting up a DOS filesystem

The **Dosfsys** filesystem manager provides access to DOS files and directories that reside on DOS disks or partitions. **Dosfsys** can support up to eight drives.

You can create, read, write, and delete files and directories on DOS disks with most QNX programs and standard QNX utilities (such as **mkdir**, **ls**, and **rmdir**). Most QNX utilities will work with DOS files, provided the DOS file structure allows for the functionality required by the utility.

Your own C programs will also be able to process DOS files just as they process QNX files, by using the standard I/O functions such as *open(), read(), write(), close(), seek(),* etc. When you read DOS directories, they'll be presented to you in QNX format.

Invocation modes

Dosfsys has four invocation modes:

Dosfsys [-S|-s] [-e] [-m] [-t] [*dos_drive***=***qnx_drive***[,R]]... &**

Dosfsys -i [-n *node***] [***dos_drive_path***]...**

Dosfsys -o [-n *node***]**

Dosfsys -x [-n *node***]**

The **-i** option lets you get information about the currently adopted DOS drives. The **-o** option lets you see the names of any files currently open on each device. The **-x** option terminates the **Dosfsys** server.

If you don't specify **-i**, **-o**, or **-x**, **Dosfsys** will start up and try to adopt the specified drives.

To start or terminate the **Dosfsys** server, you must be logged in as the superuser (**root**).

Starting Dosfsys

When you start **Dosfsys**, it:

- opens the specified drive(s)
- adopts the root DOS name (**/dos**)
- registers the name **qnx/dosfsys** with the local Process Manager

If no options have been specified, or if either **-S** or **-s** has been specified, **Dosfsys** scans the **/dev** directory for valid DOS drives to adopt. Unless otherwise specified on the command line, **Dosfsys** will adopt up to eight drives.

DOS primary partitions (**/dev/hd0t1**, **/dev/hd0t4**, **/dev/hd0t6**, etc.) and extended partitions (**/dev/hd0t1.1**, **/dev/hd0t1.2**, etc.) are mounted starting at drive C (**/dos/c**).

Floppy drives are mounted as follows:

```
/dev/fd0   →  /dos/a
/dev/fd1   →  /dos/b
```

Dosfsys name adoption

The **Dosfsys** filesystem manager can adopt up to eight drives. As mentioned above, **Dosfsys** will adopt the name **/dos** as a system prefix. It will also manage each specific drive's name as **/dos/a**, **/dos/b**, etc. These names aren't registered in the system prefix tree but are kept internally by **Dosfsys**. This will be transparent to the user except for the fact that the user won't be able to create files or directories at the **/dos** root.

DOS devices

A DOS device could be one of the following:

- a DOS partition on a hard disk
- a floppy diskette
- an image of a DOS partition or diskette

To create an image of a DOS diskette or DOS partition, you use the QNX **cp** utility. For example, to copy an image of a DOS floppy in your floppy drive 0, you could use the following:

```
cp /dev/fd0 /usr/qnx/dosa
```

and then invoke **Dosfsys** as follows:

```
Dosfsys a=/usr/qnx/dosa &
```

The same could be done with a hard disk partition. **Dosfsys** will handle these images just as it would the actual device.

For all *non-removable* devices, **Dosfsys** immediately reads the DOS Boot Parameter Block (BPB) as well as part of the File Allocation Table (FAT) at startup. For *removable* devices, the BPB and the FAT are read only when the drive is being accessed.

When **Dosfsys** has a *non-removable* device open, the device is locked for WRITE so no other process can write to this device without going through **Dosfsys**. Removable devices are kept open and locked only during accesses (e.g. during reading or writing to the disk). Note that unless you specify the **R** option, all drives have READ/WRITE access.

DOS version support

The **Dosfsys** manager supports all DOS partitions formatted with DOS 2.1 or later, including standard primary DOS partitions, DOS large partitions (DOS 4.0 and DOS 5.0 **>** 32M), and DOS extended partitions (type 5). Hard disks, $5\frac{1}{4}''$ floppies, and $3\frac{1}{2}''$ floppies are supported.

DOS partition types

These are the standard DOS hard disk partition types:

Type:	Description:
1	DOS primary partition (12-bit FAT)
4	DOS primary partition (16-bit FAT; ≤ 32M)
5	DOS extended partition (DOS 3.3 or later)
6	DOS primary partition (DOS 4.0 or later; **>** 32M)
11	DOS 32-bit FAT; partitions up to 2047G
12	Same as Type 11, but uses Logical Block Address **Int 13h** extensions
14	Same as Type 6, but uses Logical Block Address **Int 13h** extensions
15	Same as Type 5, but uses Logical Block Address **Int 13h** extensions

DOS text files

DOS uses a structure for text files that's different from the one used in QNX (by text files we mean line-oriented files containing lines of ASCII text separated by line-separator sequences). In DOS, each line of a text file is terminated with a carriage return/linefeed sequence (CR/LF); in QNX each line is terminated by a linefeed character (LF).

The **Dosfsys** manager doesn't translate these files. All files are treated "as is." Therefore, you may need to use the QNX **textto** or **tr** utility to convert your text files before copying them to or from QNX and DOS disks.

Note also that the text files created by some DOS programs may contain a SUB character (**^Z**) as the last character of the file. This is also treated as is.

DOS binary files

Since **Dosfsys** doesn't translate the contents of files, binary files may be copied to or from the QNX/DOS partitions as is.

QNX-to-DOS character and name mapping

In DOS, your filenames are limited to the "8.3" convention (QNX has no such limitation). Also, you can't include any of these characters in a filename (although they're perfectly valid in QNX):

/ \ [ ] : * | + = ; , ?

Nor can you use these filenames:

- **AUX**
- **CLOCK$**
- **COM1**
- **COM2**
- **COM3**
- **COM4**
- **CON**
- **LPT1**
- **LPT2**
- **LPT3**

- **NUL**
- **PRN**

If you attempt to create a file that contains one of the invalid DOS characters or that has an invalid filename, you'll be denied access.

Since all DOS filenames and filename characters are allowed under QNX, no validation is required on these filenames.

DOS also maps all alphabetical characters to uppercase, so **Dosfsys** maps these characters to uppercase when creating a DOS filename; it maps a filename to lowercase when returning the filename to a QNX application.

Translating QNX filenames

If you specify **-t** when starting **Dosfsys**, the server will translate and/or truncate any QNX filenames that don't conform to the DOS naming conventions. The following table describes how names are translated:

If the QNX filename:	Dosfsys will:
starts with a dot (.)	change the . to a **$**
contains multiple dots	change all but the last dot to a **$**
has a prefix longer than eight characters	truncate the prefix to eight characters
has a suffix longer than three characters	truncate the suffix to three characters

The Microsoft Windows 95 release has added support for long filenames that aren't limited to the traditional DOS 8.3 filename convention. **Dosfsys** can read, but not create, long Windows 95 filenames. To have **Dosfsys** recognize long Windows 95 filenames, specify the **-L** option when starting **Dosfsys**.

Here are some examples:

QNX filename:	DOS filename:
`.profile`	`$profile`
`a.b.c.d`	`a$b$c.d`
`longfilename`	`longfile`
`longfilename.extension`	`longfile.ext`
`a..b..c.def.g.h`	`a$$b$$c$.h`

DOS volume labels

DOS uses the concept of a volume label, which is an actual directory entry at the root of a DOS filesystem. To distinguish between the volume label and an actual DOS file, **Dosfsys** places an equal sign (**=**) as the first character of the volume label name. **Dosfsys** treats this directory entry as a zero-length, read-only file whose permissions cannot be changed.

DOS-QNX permission mapping

DOS doesn't support all of the permission bits that QNX does. The DOS attribute bits are as follows:

- READ_ONLY
- HIDDEN
- SYSTEM

- VOLUME_LABEL
- DIRECTORY
- ARCHIVE

Attribute-bit translations

Dosfsys uses the following mapping logic to handle the QNX-to-DOS attribute-bit translations:

- If the entry is a directory, set the DOS DIRECTORY file bit
- If the entry is a file, and if all the QNX WRITE bits are off, set the DOS READ_ONLY bit

Permission-bit translations

The following mapping logic is used to handle the DOS-to-QNX permission-bit translations:

- Set the QNX READ permission bits for user, group, and other.
- If the entry isn't a volume label, and if the entry isn't read-only, set the QNX WRITE permission bits for user, group, and other.
- If the entry is a directory, set the QNX DIRECTORY and EXECUTE bits for user, group, and other.
- If the entry is a file, set the QNX REGULAR FILE bit.

If a file is written to, the DOS ARCHIVE bit will be set.

If **Dosfsys** is started with the **-e** option, all DOS files and directories will have the QNX EXECUTE bits set for group/owner/other.

File ownership

Although the DOS filesystem doesn't support user IDs and group IDs, **Dosfsys** will not return an error code if an attempt is made to change the group ID or user ID with the **chown** utility or *chown()* library function. An error isn't returned because a number of utilities make

use of the *chown()* library function, which could result in many error messages being displayed.

All files under **Dosfsys** are owned by the superuser (uid=0, group=0) with access to all.

Terminating Dosfsys

The **-x** option terminates the **Dosfsys** server. If you specify **-x**, no new *open()* requests will be accepted and the server will terminate once all active files (i.e. files still open) are closed.

Error codes returned by Dosfsys

If a request made to **Dosfsys** isn't supported, the EOPNOTSUPP error code will be returned to the application making the request. Examples of requests not supported by **Dosfsys** include:

- LINK
- BLOCK_READ
- BLOCK_WRITE
- MOUNT_PARTITION
- MOUNT_RAMDISK
- PIPE
- DISK_GET_ENTRY
- RECORD LOCKING
- SYMBOLIC LINK

If **Dosfsys** detects a corrupt filesystem, it will return EBADFSYS, at which point you may wish to run the **CHKDSK** utility under DOS to correct the problem.

The DOS filesystem structure is such that the root directory's size is fixed at format time and cannot be resized. If it does become full, an error will be returned (ENOSPC).

Chapter 4

Connecting Character Devices

In this chapter. . .

Starting device drivers

A QNX system will usually contain one or more *character devices*. All such devices are managed by the **Dev** process. This process must be started first (as **root**) before any device drivers are started:

```
/bin/Dev &
```

Once **Dev** has been started, one or more of the following device drivers may be started (as **root**):

- **/bin/Dev.con** (QNX console device driver)
- **/bin/Dev.par** (parallel printer device driver)
- **/bin/Dev.ser** (serial device driver)
- **/bin/Dev.pty** (pseudo tty driver)

Each of these drivers is described in more detail in the *Utilities Reference*.

You may need to modify the **Dev** command in your **sysinit.***node* file to support a larger number of devices than the default. For example, if you specify:

Dev [*any_other_options***] -n 64 &**

then **Dev** will support 64 terminal devices (virtual consoles, serial terminals, and pseudo tty devices). For more information, see **Dev** in the QNX 4 *Utilities Reference*

Parallel devices

Parallel ports are normally used to communicate with parallel printers. Apart from starting up the driver, there's little work for you to do other than connect the printer to the machine. If your printer's on a network, see the Print Spooling chapter to set up the printer as a shared resource.

Single parallel port

If only one parallel port is available on a machine, then no parameters are required:

```
Dev.par &
```

Started this way, the parallel driver will create a device called **/dev/par1**, which corresponds to the first parallel port found by the BIOS (LPT1).

Multiple parallel ports

If your machine has more than one parallel port, then you'll need to start an additional **Dev.par** for each extra port. You must provide a unique name for the extra devices. For example:

```
Dev.par &
Dev.par -b 2 -N laser &
```

These commands will create a device called **/dev/par1** on LPT1, and a second device called **/dev/laser** on LPT2.

Output buffers

If you have the memory, you might find that specifying large output buffers significantly improves turnaround time when sending data to your printer. Here's an example of a parallel device created with a 30K output buffer:

```
Dev.par -O 30000 &
```

Testing parallel drivers

If you're having problems with your printer, you can test the basic operation of the driver using the **cat** utility. The **cat** utility writes the contents of a file to standard output. You can redirect the output to the printer to determine whether the parallel driver is working as expected:

```
cat myfile > /dev/par
```

For many printers, you'll need to send a formfeed (Ctrl – L) to cause the page to print.

Serial devices

In this section we'll look at serial hardware, the RS-232 protocol, and various configuration issues.

Hardware adapters

I/O addresses

The **Dev.ser** driver can support one or more serial ports. The hardware interface to the computer consists of a Universal Asynchronous Receiver/Transmitter (UART) for each serial port. The driver supports the 8250, 16450, and 16550 families of serial controllers.

Each UART uses eight consecutive addresses in the I/O address space of the computer. The **Dev.ser** driver is informed of the I/O address range for each UART by command-line arguments when it is started.

Hardware interrupt

Just as important as the I/O address is the hardware interrupt generated by each UART. Most PCs provide several hardware interrupt signals on the bus, labeled IRQ2 through IRQ15 (except for interrupts 8, 9, and 13, which are used internally by the system motherboard).

These interrupt signals are active-high TTL logic signals on ISA buses, which means that you can connect only *one* adapter card to any one interrupt signal.

Serial adapter cards come in various configurations. Adapter cards with only one serial port typically offer only a limited set of choices for I/O address and hardware interrupt. The following table shows some commonly found combinations, but we recommend that you

read the hardware documentation carefully to discover the choices available on a given manufacturer's adapter card:

Name:	Address:	Interrupt:
COM1	**3F8**	IRQ4
COM2	**2F8**	IRQ3
COM3	**3E8**	varies
COM4	**2E8**	varies

Multiport serial adapters

You can usually configure multiport serial adapters to respond to a wide range of I/O addresses. The adapters may also give you considerable flexibility in selecting hardware interrupts. A good choice for I/O addresses is often the 280-through-2BF range.

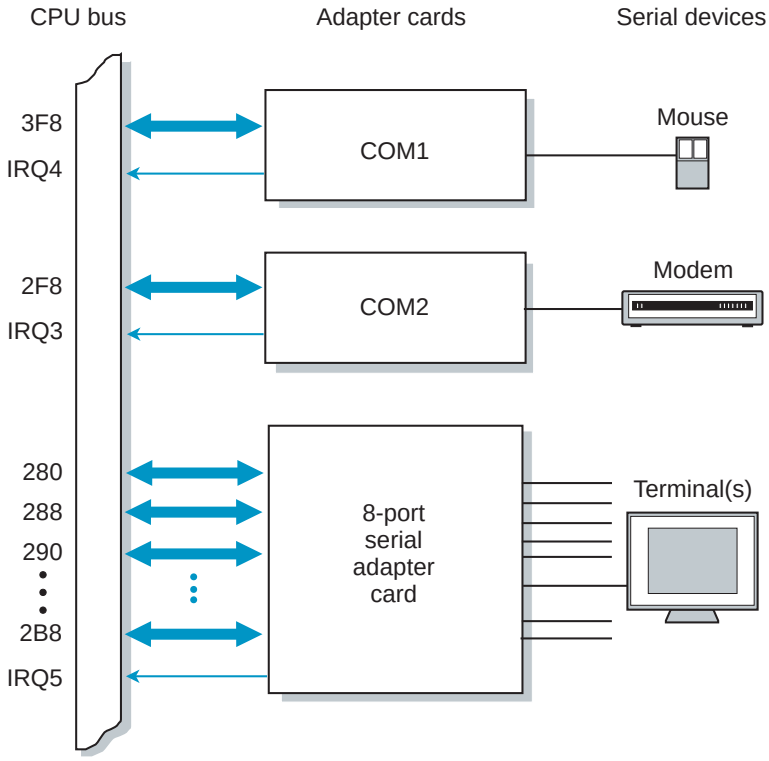
☞ Note that this discussion applies only to "dumb" serial cards. "Smart" cards are typically supported by drivers supplied by the hardware manufacturer — refer to the docs that came with the card.

Because of the limited number of hardware interrupts available, these cards will often "OR" the interrupt lines from the individual UARTs into a single interrupt connected to the bus.

☞ QNX allows for many serial ports to share the same interrupt, since **`Dev.ser`** will check every UART that shares an interrupt.

Typical hardware installation

The following diagram shows a sophisticated configuration of serial adapter cards in a QNX system.



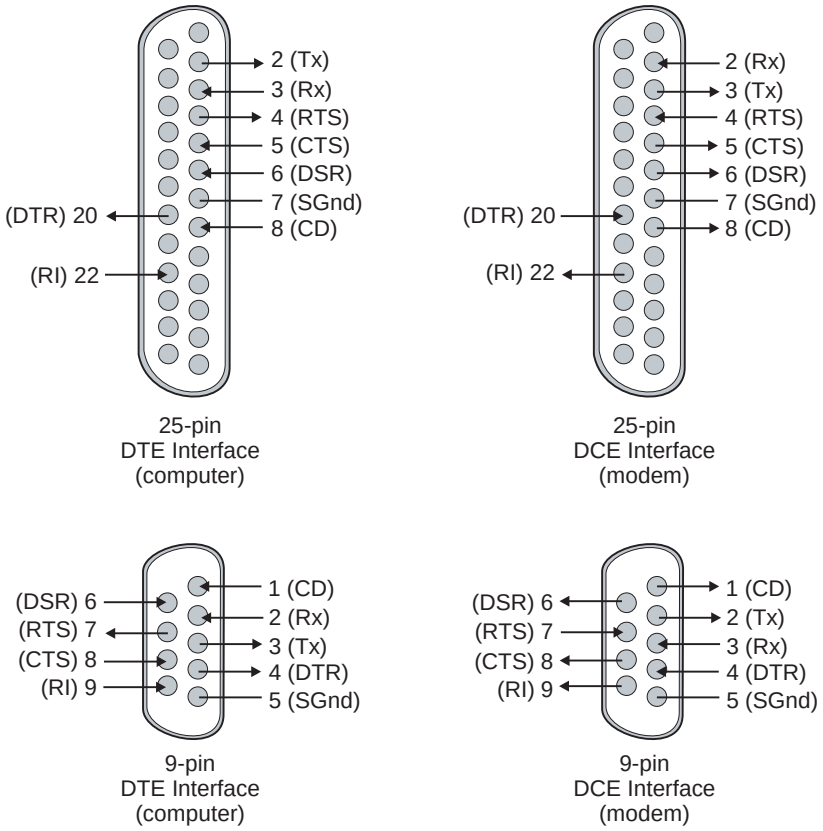
☞ For proper operation, each serial channel must have a unique I/O address, and each adapter card must use a unique hardware interrupt.

The RS-232 serial protocol

The RS-232C asynchronous communications protocol defines the electrical and physical interface between Data Terminal Equipment (DTE or *terminals*) and Data Communications Equipment (DCE or *modems*).

Electrical interface

The following figure shows the cabling assignments of an RS-232 connection.



The host computer is usually configured as a DTE, acting as a terminal device. We assume that the computer will be connected to a modem.

The RS-232 signals have the following names:

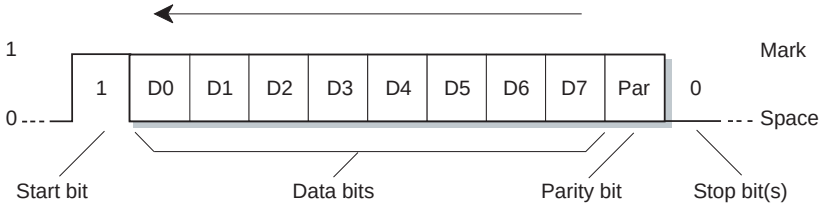
Signal:	Meaning:
Tx	transmit data
Rx	receive data
RTS	request to send

continued...

Signal:	Meaning:
CTS	clear to send
DSR	data set ready
DTR	data terminal ready
CD	carrier detect
RI	ring indicator
SGnd	signal ground

Serial protocol

Data is transmitted asynchronously using a bit protocol as shown below:



Normally, an RS-232 data line is in the SPACE (0) condition. A transmitted character consists of bits in the following order:

- START bit (always 1)
- 5 to 8 data bits (least-significant bit first)
- parity bit (optional)
- one or more STOP (0) bits

The duration of each bit is defined by the *baud rate*, which indicates the number of bits per second that can be transmitted.

If you use the parity bit, it can be one of:

odd sum of data bits plus parity bit is odd

even	sum of data bits plus parity bit is even
mark	always 1
space	always 0

Session control

RS-232 uses the DTR and DSR lines to control communication sessions. The terminal raises DTR when it is powered up and available. Similarly, the modem raises DSR when it is powered up and available (but not necessarily *connected* to a remote modem). No communication is expected to occur unless both DTR and DSR are raised.

A terminal indicates that it no longer wishes to communicate by dropping the DTR line, which causes most modems to hang up the telephone line, thus releasing the connection.

A modem indicates that it has established a connection by raising CD. Some modems will also indicate that they have detected (but not yet answered) an incoming call by raising RI.

Flow control

The RTS and CTS lines control the flow of data between terminal and modem. The terminal raises RTS when it is capable of receiving data on the Rx line. Similarly, the modem raises CTS when it can accept data on the Tx line.

Configuring serial ports

You use the **stty** utility to set the four major parameters that define an RS-232 link.

Data bits

QNX supports four character sizes. You choose the size of data character with the following **stty** command:

stty bits=*n*

where *n* can be 5, 6, 7, or 8 (the default).

This parameter defines how many bits following the start bit will be used to form the character.

Stop bits

It's possible to transmit data that is followed by either one or two stop bits. Two stop bits are used only to slow down the overall transmission of data so that the remote end has a chance to keep up. Using **`stty`**, you specify one of these:

`stty stopb=1` (the default)

OR

```
stty stopb=2
```

Parity

To disable the transmission of parity bits and suppress the checking (in hardware) of received parity bits, you specify:

```
stty par=none
```

Parity is disabled by default.

If parity is used, you specify one of the following values:

```
stty par=odd
stty par=even
stty par=mark
stty par=space
```

Baud rate

You can specify the baud rate with the **`baud=`***number* option of the **`stty`** utility. For example:

```
stty baud=2400
```

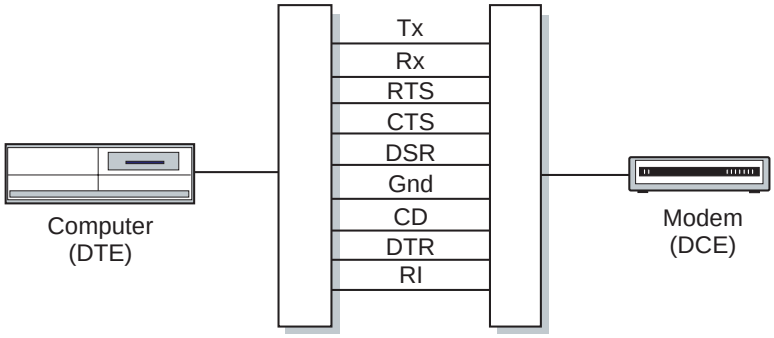
The **Dev.ser** serial driver defaults to 9600 baud. Third-party drivers may have different defaults.

Connecting serial devices

High-speed ECC modems

High-speed, error-correcting modems are becoming very sophisticated — they work best when all hardware handshaking signals are respected. These modems often communicate with the host computer at a fixed high-speed baud rate (e.g. 115200 or 57600 baud) and use the RTS/CTS handshaking lines to regulate the actual flow of data over the communications link. QNX is ideally suited for communicating with such modems.

A straight-through cable is used to connect the modem to the computer.



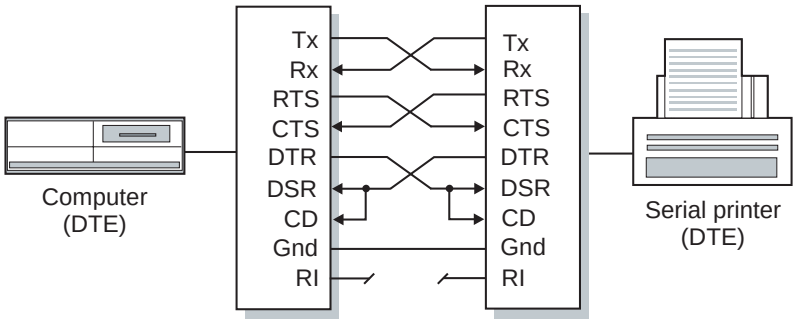
Straight-through cable

Serial printers

Serial printers are usually *bidirectional* devices. Data flows from computer to printer as expected, but since printers can't keep up with the host computer, serial printers often use software flow control to regulate the flow of data. In other words, they transmit XON and XOFF characters back to the computer. Some printers use the

hardware handshaking lines for this purpose, some support both forms of flow control.

To be safe, you should connect all nine signals, although printers that support only software flow control may function just as well with a three-wire cable (Rx, Tx, and Gnd). Also, since printers are usually configured as Data Terminal Equipment (DTE) — just like the host computer — you may need to use a null-modem cable.



☞ The actual wiring of the CD and RI pins may vary depending on the manufacturer's cable.

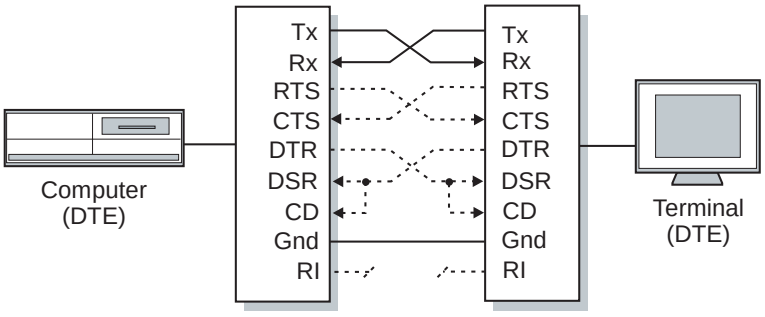
If the printer uses:	You use:
software flow control	**`stty +osflow </dev/ser1`**
hardware flow control	**`stty +ohflow </dev/ser1`**
both software and hardware flow control	both **`stty`** options

☞ If your printer's on a network, see the Print Spooling chapter to set up the printer as a shared resource.

Terminals

Terminals operate with or without flow control and at a fixed baud rate. Unlike printers, terminals can usually keep up with the host computer at a supported baud rate. A simple three-wire cable may be sufficient, but we recommend a nine-wire cable.

Like the host computer, terminals are normally configured as DTE devices, so a null-modem cable is usually required.



Null-modem cable

If your terminal supports software flow control (e.g. VT100), it's a good idea to enable output software flow control and lock it on:

```
stty +osflow +lksflow </dev/ser1
```

You could also use the **-x** option when starting **Dev.ser**.

Configuring serial lines for terminals and users

QNX can be used as a full-function time-sharing system. Many users can be connected to some or all of the computers in a QNX network via hardwired terminals or through modem links to terminals at remote sites.

Assuming that the terminal/modem is properly configured, you'll also need to provide a mechanism for users to login.

Logging in

Assume a terminal, properly connected and configured, on the serial port **/dev/ser1**. The simplest way of allowing a user sitting at that terminal to log in is to use this command:

```
on -t /dev/ser1 login
```

The user will be able to log in and execute commands. However, when the user logs off, the shell session is terminated and the user won't be able to log back in. To give users a way to log in again, specify the **-t** option to the **tinit** utility when the terminal is initialized:

```
tinit -t /dev/ser1 /dev/ser2 &
```

Automating logins

You can use the **tinit** utility to automate the login process. When you start **tinit** with the **-T** option, this utility waits for a character to be sent from the terminal. You can specify (via the **-c** option) a command for **tinit** to execute when a key is pressed. By default, **tinit** starts the **login** utility.

After the user logs off, **tinit** waits once again for a character to be sent. The following command starts **login** on two serial devices (**/dev/ser1** and **/dev/ser2**) when a key is pressed:

```
tinit -T /dev/ser1 /dev/ser2 &
```

Launching custom applications

You can start up any program using the **-c** option to **tinit**. You can even specify a different program on each device. In some environments, a "canned" application is always expected on a given terminal.

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

Modem access

The **modem** utility is provided to respond to modems. Used in conjunction with **tinit**, the **modem** utility can provide excellent dial-up capabilities. A typical dial-up system using QNX might have several serial ports (**/dev/ser1**, **/dev/ser2**, etc.) and might use the following command to permit dial-up access:

```
tinit -c "modem -b 38400 -L" -t /dev/ser1 &
```

The **tinit** utility can launch **modem** on each of the serial lines. When communication is established, **modem** will:

- answer the phone
- determine and set the proper baud
- *exec* into **login**

When the user either logs off or hangs up, **tinit** will once again launch a new **modem**, which will wait for another call. For more information, see **modem** in the *Utilities Reference*.

Exclusive access to serial devices

Multiple processes can open the same device; if they read from the device, their reads will be interleaved. This is generally undesirable. To keep this from happening, QNX maintains an “open count.”

The open count, like locks, is advisory only. Programs that require exclusive use of a device should check the open count (through the *dev_info()* routine) before reading from the device. An open count of one would indicate that your program is the only one with the device open. The **modem** and **qtalk** utilities make use of this facility so that you don’t have to dedicate a serial line for incoming calls.

The **qtalk** terminal emulator checks if the line is free, and if so, will connect underneath **modem**, allowing you to place an outgoing call. When the **qtalk** session in complete, **modem** will again be in control, waiting for incoming calls to arrive. For more information, see **qtalk** in the *Utilities Reference*.

Testing serial drivers

If you're having problems with a serial line, you can use the **qtalk** utility to verify that the serial driver is working as expected. Running **qtalk** lets you communicate with a connected system through a serial line. With **qtalk** running, you can relay the characters you type through the serial link.

The following command would start a **qtalk** session with the system connected directly to a device:

qtalk -m *device*

Once in **qtalk**, your keystrokes will be sent through the serial line, and incoming data on the serial line will be displayed on your screen. If this doesn't happen, there could be a problem with the serial port configuration. To test for basic serial port functionality, connect a modem to the serial line and try to talk to the modem in its command mode.

Troubleshooting serial device problems

The following table describes the actions you can take if you encounter some of the common problems involved in connecting serial devices.

Problem:	Probable cause:	Remedy:
No data can be sent or received	Wrong cable	Check cables; use null-modem if necessary
	Wrong I/O ports	Check hardware settings and verify correct parameters to **Dev.ser**
	Interrupt conflict	Change interrupt on adapter card

continued...

Problem:	Probable cause:	Remedy:
Data is received and transmitted only when another serial port is in use	Interrupt conflict	Check hardware interrupts and **Dev.ser** startup parameters; make sure that two serial adapters are *not* using the same IRQ
Data is lost occasionally	Flow control supported, but not enabled	Determine the type of flow control supported by the device and enable with **stty** (**ihflow**, **ohflow**, **isflow**, and **osflow**)
	Flow control not supported	Reduce baud rates and/or increase stop bits; if only received data is lost, specify larger input buffer to **Dev.ser** (**-I** option)
	Cabling problems	Make sure cable is well grounded and not too long; also, verify that all RS-232 wires in the cable are connected
Data characters are unrecognizable	Wrong baud rate or parity	Use **stty** to set the correct baud rate and/or parity
Some characters are fine, some aren't	Wrong parity	Try different parity using **stty**

Pseudo terminal devices

The **Dev** manager is responsible for managing all terminal devices, including "pseudo" terminal devices or ptys. Pseudo devices are allocated in pairs; the members of each pair are internally connected

to each other. Any data written to the master side (e.g. **/dev/ptyp0**) will be available for reading on the slave side (e.g. **/dev/ttyp0**).

You may run into the common problem where **Dev** hits its maximum number of devices and rejects **Dev.pty**'s attempt to register. If this happens, increase the value of **Dev**'s **-n** option.

For example, to create 16 device pairs called **/dev/ptyq0** through **/dev/ptyqf** and **/dev/ttyq0** through **/dev/ttyqf**:

```
Dev.pty -n 16 -l q &
```

Console Devices

The display adapter, the screen, and the system keyboard are collectively referred to as the *console* in QNX. To let you interact with several applications all at once, QNX permits multiple sessions to be run concurrently on consoles by means of *virtual consoles*. These virtual consoles are usually named **/dev/con1**, **/dev/con2**, etc.

Dev manages consoles via its associated console I/O drivers **Dev.con** or **Dev.ansi**. Again, **Dev** itself must be running *before* any of its drivers.

To set the number of consoles for a machine, you use the console driver's **-n** option. For example, the following command starts the **Dev.con** driver with a maximum of 6 virtual consoles:

```
Dev.con -n 6 &
```

You can switch from one console to the next by using this simple keychord:

Ctrl – Alt – Enter

For more information, see the appendix on "QNX Console & Keyboard Conventions."

Chapter 5

Licensing

In this chapter. . .

Operating system licensing

QNX must be licensed for each computer it runs on. This is true whether you're using 10 standalone machines or 10 networked machines.

In a network, each machine is referred to as a "node" and each node is assigned a "logical" node ID. Logical node IDs range from 1 to the number of nodes in the network. For example, a network licensed for three nodes would support logical node IDs 1, 2, and 3.

The number of nodes in your network is determined by the number of licenses you've installed. You install old-style licenses by running the **`license`** utility or new-style licenses by adding them to the **`/.licenses`** file (see the "Adding licenses" section).

For licensing to work across a network, the boot server (the machine with the licenses) must be running the **`nameloc`** utility. If a machine boots from local hard disk (with licenses) but isn't on a network — a portable, for example — it must also run **`nameloc`** for licensing to work.

Old- and new-style licenses

QNX now supports two licensing schemes concurrently: old-style (earlier than QNX 4.23) and new-style (QNX 4.23 and later). All QNX 4 products released prior to QNX 4.23 use the old-style licensing scheme.

Old-style

Old-style licenses are kept in the **`/etc/licenses`** directory. There's one file per license, and they're always distributed on disk. Old-style licenses can be installed and transferred from disk to disk only by using the **`license`** utility.

New-style

New-style licenses are kept in the **/.licenses** file. They can be distributed on disk, but in most cases the license numbers are recorded on paper as a license certificate. New-style licenses on disk must be installed using the **license** utility, which appends them to the **/.licenses** file. You may add new-style licenses on certificate to the **/.licenses** file using any text editor. The **/.licenses** file can be transferred from disk to disk using the **cat** command (see the section on "Transferring a license to another node" in this chapter).

The /.licenses file

New-style licenses are maintained in the file **/.licenses**. The following shows the contents of a sample (i.e. bogus) **/.licenses** file. It contains licenses for six products, including the QNX OS:

```
  qnx.00090209-02lg-0947-48g2-00p7-0044 (4 node)
  qnx.00035882-02lg-0947-48g2-00p7-0044 (4 node)
  wcc.00375634-0l04-4k0l-0x6l-6112-5409 (4 node)
 phab.00006233-0040-0527-00l4-ji3g-1130 (4 node)
 phrt.00006932-007l-8070-g140-l410-84n3 (4 node)
 xrun.00004746-0l04-4l0k-0x7o-5514-8609 (4 node)
motif.00053489-00lk-0245-44e9-04i4-0004 (4 node)
```

The *product_name.nnnnnnnn* item at the beginning of each line represents the serial number that belongs to the specified product.

All QNX licensing is done on a per-node basis. In the example above, two QNX licenses are installed, and four concurrent QNX sessions can be run from each. This means that a maximum of eight copies of QNX can be running concurrently. If you try to run a ninth node, it won't be able to talk to the network.

The etc/licenses directory

Old-style licenses are kept in the **/etc/licenses** directory. Each license is an individual disk file that can be viewed using the **ls** command:

```
ls /etc/licenses
```

QNX reports old-style license information in the following format:

```
qnx0000178n004
```

where “qnx” is the name of the product, “0000178” is the serial number, “n” is a separator, and “004” means that the file is for four nodes. If you installed other QNX applications (e.g. Watcom C), their licenses will also be displayed.

Verifying your licenses

The **licinfo** utility lets the superuser list all licenses that have been installed. You can specify command-line options to display information by node or by a particular product:

licinfo (lists all licenses in use in the network)

licinfo -a (lists all licenses whether they’re in use or not)

licinfo -l wcc (lists all C compiler licenses)

licinfo -n 1 (lists all licenses on node 1)

Adding licenses

In a network, the licenses must be installed on the boot server. If you wish to boot some machines from their own hard disks, you must install all licenses on all those machines that will boot from hard disk. The newly installed licenses must be activated afterwards in order for QNX to honor them (see “Activating licenses” below).

New-style licenses

When you buy new software, your license(s) are supplied either on diskette or on a license certificate. Licenses on diskette are added to the **/.licenses** file automatically during the installation process. If you received a certificate, it will have new license number(s) on it. You’ll need to add these license numbers to the **/.licenses** file manually using a text editor.

Don't forget to activate the licenses on your boot server and on all nodes that boot from that server using **license -r**.

Old-style licenses

The following command installs the license from floppy drive 0 to your hard disk. Note that the floppy driver (**Fsys.floppy**) must be running:

```
license
```

Activating licenses

After you install a license (typically on the boot server), you must activate it. To do so, run **license -r** on the node (or boot server), which reads all licenses from **/.licenses** and **/etc/licenses** (if it exists) and places them in the machine's in-memory license database.

Nodes that boot over the network will inherit the licensing database from their boot server. You can run the following command to refresh a node's license information from the boot server's in-memory license database:

license -R *boot_server_node_ID*

Transferring a license to another node

Old-style licenses can be transferred from one location (or node) to another using the **license** command. For example, the following command would transfer all licenses from **/etc/licenses** on node 61 to **/etc/licenses** on node 71:

```
license //61/etc/licenses //71/etc/licenses
```

New-style licenses are kept in the **/.licenses** file. To copy this file from node to node, use **cat** (not **cp**) so that you won't inadvertently

overwrite the contents of a file. For example, the following command will *append* the contents of node 1's **/.licenses** file to node 5's **/.licenses** file:

```
cat //1/.licenses  >>//5/.licenses
```

CAUTION: You should protect your **/.licenses** file so that the system administrator is the only user with read-write access to the file:

➤ As **root** enter this command:

```
chmod 600 /.licenses
```

Chapter 6

Networking

In this chapter...

Introduction

QNX is a network-distributed operating system. Each computer QNX runs on is called a *node*. A single computer can be considered a one-node network.

The QNX Network Manager recognizes a wide range of protocol packets and passes them to their appropriate destinations transparently.

If you need full TCP/IP-based network communications between a QNX machine and a non-QNX machine, you'll have to purchase and install TCP/IP for QNX.

Logical node IDs

All QNX nodes are assigned a unique logical node ID represented by a positive integer. The operating system uses these identifiers to communicate with other nodes on a network.

You should try to assign logical node IDs in an uninterrupted sequence starting at 1 (e.g. 1, 2, 3, 4, 5, ...). There are two reasons for this:

- QNX licensing determines the total number of logical nodes in the network. A five-node network license allows nodes 1 through 5 to communicate. A node 6 would be isolated and ignored, even if one of nodes 1 through 5 were omitted.
- A few important utilities (e.g. **`nameloc`**) attempt to communicate with each node of the network in turn. They do this by sending messages starting at node 1 and continuing up to node *n*, where *n* is the number of installed network licenses.

Logical network IDs

Nodes are connected via one or more physical networks. Each network in an installation is a separate communications link.

Networks, like nodes, are assigned logical network IDs that must be unique.

A single network (the most common case) will default to logical network ID 1. If you have a second network, assign it network ID 2.

Physical node IDs

Although QNX processes deal with logical node IDs, the network cards themselves communicate with physical node IDs. This physical ID is typically contained within the network card itself. The format of the physical ID depends on the network type (e.g. Ethernet, Arcnet, Token Ring) and is invisible to most applications. Physical IDs must be unique within a network, but they don't have to be unique across all networks in an installation.

Boot servers and booting nodes

A QNX LAN is essentially a peer-to-peer network that you set up in two phases:

- configuring the boot server
- configuring the booting nodes

Any QNX node that has a hard disk may boot from its own hard disk. A boot server is simply a node that services boot requests from other nodes on a network. Usually a node is chosen to be a boot server because it's rarely rebooted.

When a node boots from the network, it obtains an OS image from the boot server. This boot image is loaded into memory at the booting machine, which typically inherits the following parameters from the downloaded boot image:

- the time
- the global list of symbolic process names (see **`nameloc`** in *Utilities Reference*)
- its licensing capabilities

You can set up your network with multiple boot servers to provide fault-tolerance (in case your primary boot server becomes inoperative) and to distribute the boot resources to avoid boot bottlenecks (aka “bootlenecks”).

How network booting works

Network cards in a booting node issue boot requests by either:

- sending a boot request to a specific boot server

  OR

- broadcasting a boot request to all nodes, expecting that a boot server will respond to the request

The QNX Arcnet card, which has nonvolatile RAM that can store the node ID of the server, uses the specific boot method. Most other network cards such as Ethernet and Token Ring don’t have the means to store the physical node ID of the server, so they must use the broadcast method.

Network bridging between QNX LANs

Any QNX network node can act as a bridge between two different QNX networks, as long as they’re both IEEE 802-type networks. For example, QNX network bridging lets you connect an Ethernet network to a Token Ring network or to an FDDI network. This is possible because QNX uses the same packet format and protocol on all IEEE 802-based networks. All you have to do is connect two different networks to the same node. The node can be a boot server or simply a booting machine.

QNX can’t create a bridge to an Arcnet network.

Planning your network

One network or more?

If you're setting up a small network, it will probably need only a single boot server, typically node 1. But if you're setting up a larger network that will span several departments, you should set up a boot server for each department. The users of each department will still be able to access files (subject to file permissions) in other departments.

For example, let's assume you have three departments: R&D, Marketing, and Operations. You may elect to have each department boot from its own boot server, which could also act as the department's major file server.

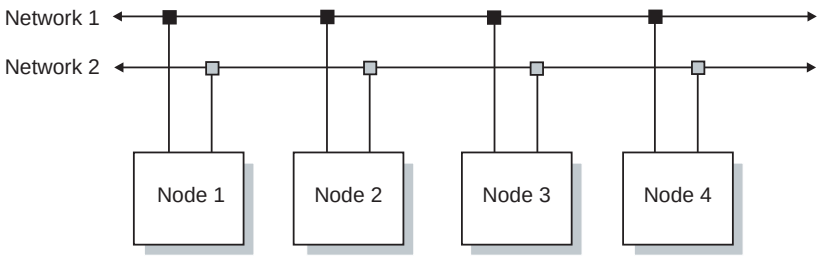
This setup creates a fault-tolerant environment — a failure in one department won't stop another department from booting. For large numbers of nodes, this practice should also reduce the potential for a boot bottleneck when people in the office start work first thing in the morning.

Multiple network links

As described in *System Architecture*, having more than one network link in a computer provides fault-tolerance and increased throughput.

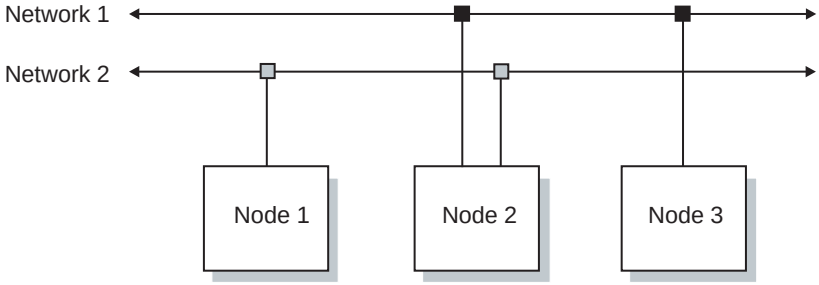
Communicating across networks

In QNX, a node can communicate with multiple networks simultaneously. Consider the following diagram where four computers are linked via two networks:



In this fault-tolerant configuration, each of the four nodes can communicate directly with any other node through either network 1 or network 2.

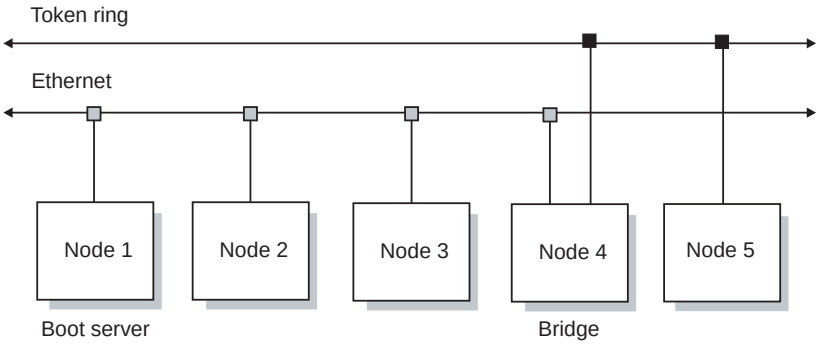
Compare this to the following setup, where nodes 1 and 3 are connected to separate networks, and node 2 is common to both networks. Nodes 1 and 3 aren't directly connected to each other, but they can both access node 2.



If node 2 were connected to two IEEE 802-type networks, QNX network bridging would automatically bridge the two networks through node 2.

Setup considerations

You may want to prepare a diagram for your network, similar to the one below. This example diagram shows a network consisting of five nodes. One of the nodes is the boot server and three are nodes that will boot from and obtain their files from the boot server. Node 5 boots from its own hard disk.



We've assigned a logical node ID to each node; the boot server is node 1. Note that node 4 will act as a bridge between the Ethernet

network (logical network ID 1) and the Token Ring network (logical network ID 2).

☞ You can't boot across a bridge. In the previous diagram, for example, node 5 can't boot from node 1.

Network cards and drivers

Setting up a network typically involves installing a network card in your computer and connecting the card via a cable to other computers that have the same type of network card. Ethernet, Arcnet, and Token Ring are all popular network types.

In addition to the network card, you need the QNX network driver that supports it. All QNX network drivers are documented in the *Utilities Reference*. Their names start with the prefix "**`Net.`**"

Cards from different vendors for the same network media (e.g. Ethernet) may require their own special drivers. But often a driver may support cards from more than one company if the cards provide a similar hardware interface.

☞ For information on the hardware installation/configuration of a network card and the physical connection between network cards, see the docs provided with the card.

Determining physical node IDs

The method you use to determine the physical node ID of a network card depends on the type of card you have. In many cases, you'll find a label with the address of the card in the box. Sometimes switches or jumpers on the card determine the physical node ID of the card.

Ethernet & Token Ring

Every Ethernet or Token Ring card is shipped with its own physical node ID built into the card. This ID, which is unique worldwide, is 48 bits long in order to conform to the IEEE 802 standard.

On Ethernet cards, the ID may be printed on a label somewhere on the card or on the box. Typically, diagnostic software is used to display the label. The address format of the ID may vary. For example, the following are all identical:

```
0000c0 4a9330
0000c04a9330
00 00 c0 4a 93 30
00:00:c0:4a:93:30
0000 c04a 9330
```

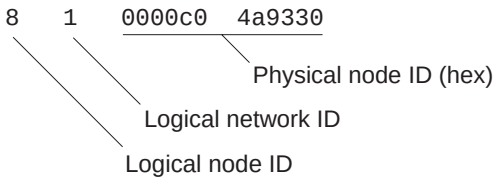
Arcnet

Every Arcnet card requires that you program a physical node ID into the card. Depending on the manufacturer, this is done by a DIP switch or is configured in nonvolatile RAM through a menu-driven interface at boot time.

The physical ID is 8 bits in length. You can't use physical ID 0, which is reserved. Since you can choose the physical node ID, we recommend that for a single Arcnet network, you make the physical node ID the same as the logical node ID. This will help keep things simple.

Assigning logical node and network IDs

Since QNX processes use logical IDs, the Network Manager (**`Net`**) must map these logical IDs to the physical IDs used by the hardware. This mapping is defined in the **`/etc/config/netmap`** file. Each line in this file defines a single-node mapping: the logical node and network IDs are followed by a physical node ID. For example, the following line would map logical node 8 on logical network 1 into a 48-bit physical ID of 0000c04a9330:



You can separate the logical and physical IDs by Space or Tab characters. The logical IDs are in decimal, while the physical IDs are in hex, unless preceded by a "**t**", in which case they're also in decimal.

For example, in the case of Arcnet, it's convenient to express the physical node ID in decimal form:

```
15   1   t15
```

The physical ID can't exceed 48 bits. The network hardware determines the number of bits needed.

Configuring a boot server

This section shows you how to set up a single boot server. Once you've set up a single server, you can set up the booting nodes. If your site requires additional boot servers, see "Multiple boot servers."

This procedure assumes that you've already done the basic installation of QNX onto your machine's hard disk as described in the Basic Installation chapter. Therefore, your machine has already been configured to boot from its own hard disk.

Turning a QNX node into a boot server involves these steps:

- installing the network card(s)
- installing your network licenses
- starting the Network Manager and the network driver(s)
- starting the **nameloc** utility
- starting the **netboot** utility
- modifying the **sysinit.***node* file
- modifying the **netmap** file
- modifying the **netboot** file

Step 1 — Install the network card(s)

Turn off the computer, then install your network card according to the instructions provided with the card. If you can, note the physical node ID of the card. When you're finished, reboot the computer.

If you weren't able to note the physical node ID when installing the card, or if the ID wasn't on the card, running **`netmap`** without any options (after the network driver starts) will display the physical node ID.

Installing multiple cards

When you place two similar cards in the same machine, don't rely on the default autodetect for I/O ports, etc. Be sure to verify that the settings are unique — you might have to specify some of this information using command-line options to the network driver. You must configure the cards such that they have no hardware conflicts.

Read the technical note for the type of card(s) you're installing before you install the card. The files in the **`/etc/readme/technotes`** directory contain useful tips, such as how to set hardware interrupts so that you can avoid potential conflicts.

Step 2 — Install your network licenses

You'll need a network license for each node in your network. When you installed QNX onto the hard disk initially, a license was installed for at least a single node. You can verify how many licenses were installed using the **`licinfo`** command. See the Licensing chapter for more information about installing and activating licenses.

Step 3 — Start the Network Manager & network driver(s)

If you told **`install`** that this node is to be a boot server, the appropriate network services will have been started already. Skip this step and go to the next one.

Starting the Network Manager & one network driver

The Network Manager (**Net**) and the network driver for your network must be running to access logical network 1.

The names for all network drivers are of the form **Net.***xxx*, where *xxx* represents the appropriate driver. For example, if you have an NE2000-compatible Ethernet card, you'd enter something similar to the following:

```
Net &
Net.ether1000 &
```

To find out which network drivers are available on your system, enter this command:

```
ls /bin/Net.*
```

Starting two or more network drivers

If you installed two network cards, you'll have to start a driver for the second network.

By default, all network drivers connect to logical network ID 1. When connecting multiple networks, you have to specify the **-l** ("el") command-line option to have a driver connect to a network other than logical network 1.

For example, the following three commands in the **sysinit.***node* file of a boot server would start the Network Manager, an Ethernet driver on logical network 1, and a Token Ring driver on logical network 2:

```
Net &
Net.ether1000 &
Net.tr164a -l 2 &
```

Note that the Network Manager must be started with the **-d** option whenever three or more network drivers are started.

This command causes **nettrap** to scan the machine it's running on for network cards:

```
nettrap
```

CAUTION: The **nettrap** utility reads and writes to I/O ports and to memory. This testing may cause unintentional side effects in other hardware residing at the tested memory and I/O addresses. If your system is set up to control external machinery, disconnect these devices before running **nettrap**.

The **nettrap** utility can also be used to start **Net** along with the appropriate network drivers (and **netmap -f**):

```
nettrap start
```

Step 4 — Start nameloc

The **nameloc** utility runs in the background and provides a network-wide naming service for all processes running under the OS. It must be running on at least one machine on the network — even if it's the *only* machine in a standalone "network" — for licensed products to work:

```
nameloc &
```

Upon starting, **nameloc** polls each node in the network for its list of global symbolic process names. For more information, see the description of the **nameloc** utility in the *Utilities Reference*.

CAUTION: The machine running **nameloc** *must* be able to talk to every other machine on the network. Make sure the machine has a *complete* **netmap** file. If you run **nameloc** on a node (e.g. a portable) that doesn't have complete network access, you'll run into all sorts of problems — wrong licensing info, flawed list of global names, malfunctioning network-wide utilities, etc.

Step 5 — Start netboot

When a booting node starts up, its boot ROM sends a boot request to its boot server or it broadcasts the boot request to any boot server that has been configured to respond. When the Network Manager on the boot server receives a boot request, it forwards the request to the **netboot** utility. To start **netboot**, type :

```
netboot &
```

When responding to a boot request, the **netboot** utility accesses the **/etc/config/netboot** file to determine which build file can be used to generate the OS image for the requesting node.

☞ Running **netboot** with the **-v** option (verbose mode) can be useful for troubleshooting. Multiple **v**'s will increase the level of info — you might consider directing the output to file or running it on its own console.

You may run **netboot** with the **-a** option to help automate network mapping (see Step 7):

```
netboot -a &
```

In this case, if the **/etc/config/netmap** file doesn't contain a mapping assignment for the requesting node, **netboot** will automatically write the required mapping to the **/etc/config/netmap** file using the next available logical node ID and the logical network ID of the network the requesting node is connected to.

Step 6 — Modify the `sysinit.`*node* file

Now that you've started all of the required network drivers and services, the node can function as a boot server. Once you've added the network startup commands to the boot server's **`sysinit.`***node* file, they'll be started automatically whenever the node is rebooted.

First, copy the boot server's **`sysinit.`***node* file (e.g. **`sysinit.1`**) to **`altsysinit`** before you continue:

```
cp /etc/config/sysinit.1 /etc/config/altsysinit
```

This saves an alternate copy of the boot server's initialization file. Now add the required entries to **`/etc/config/sysinit.`***node*. Entries similar to the following are required for a boot server:

```
Net &
Net.ether1000 &
nameloc &
netboot &
```

Step 7 — Modify the `netmap` file

The **`/etc/config/netmap`** file is the default node and network ID mapping file used by the **`netboot`** and **`netmap`** utilities. This file defines the physical node IDs, the logical node IDs, and the logical network IDs associated with each node.

An example file is shipped with QNX. The file you use must contain the logical node and network IDs for the boot server and for all the networked nodes that must communicate with the boot server. The mapping entries in the file look something like this:

Node ID	Network ID	Network Card Address
1	1	*boot_server_physical_address*

continued...

Node ID	Network ID	Network Card Address
2	1	*node2_physical_address*
3	1	*node3_physical_address*
4	1	*node4_physical_address*

Note that when you have a single network, the logical network ID is usually "**1**." All networked nodes must have a unique logical node ID. We suggest that you choose logical node ID 1 for your first (primary) boot server. In the case of a node that's connected to two or more networks, the same logical node ID will have two entries, each with a unique logical network identifier.

You can edit the **netmap** file manually, or if you started **netboot** with the **-a** option, the system will write node mappings to the **netmap** file automatically the first time you power on each node. The node you power on first will be assigned the next available logical node ID, the correct logical network ID, and the correct physical address. If you specify the **-A** option, new nodes are assigned the lowest unassigned logical node ID and the correct physical address.

If you have nodes that boot from their own hard disks, you can generate an updated "master" **/etc/config/netmap** file on node 1 as follows:

1. At all machines except node 1, create a local **netmap** file containing the logical-to-physical node mapping to node 1.
2. At all nodes except node 1, type the following command to update the in-memory network mappings on node 1:
   ```
   sin -n1
   ```
3. To save the in-memory network mappings to disk at node 1, type:
   ```
   netmap > /etc/config/netmap
   ```

If you want to propagate the mappings to other nodes, you can copy the master file from node 1. For example, to copy the master file to node 2, type:

```
cp //1/etc/config/netmap //2/etc/config/netmap
```

Mapping multiple networks

To define a node having more than one network card, you would add one entry to correspond with each network. Remember, the logical network IDs must be unique. For example, if your network had a fifth node that was connected to two different networks, you'd add two more entries to the **netmap** file:

Node ID	Network ID	Network Card Address
5	1	*node5_physical_address_card1*
5	2	*node5_physical_address_card2*

For more information about the **netmap** file, see the **netmap** utility in the *Utilities Reference*. For examples showing the mappings of multiple networks, see the "Network examples" section in this chapter.

Every time you edit the **netmap** file, you should update the in-memory network tables as well:

```
netmap -f
```

Step 8 — Modify the netboot file

When a boot server receives a boot request from a node, the Network Manager forwards the request along with the node's physical node ID and logical network ID to the **netboot** utility.

With this information, **netboot** accesses the network mapping file (**/etc/config/netmap**) to determine the node's logical node ID. The utility then accesses the **/etc/config/netboot** file to determine which build file can be used to generate an OS image for the node.

If you're using NE2000-compatible cards, your **/etc/config/netboot** file on the boot server could contain a line similar to the following:

```
* f=build/ws.ether1000
```

If you specified to **install** initially that this node is to be a boot server, your **netboot** file will already contain what you need.

You can modify this **f=** entry if you'd like QNX to use a custom build file. For example, if you copied one of the build files shipped with QNX, renamed it to **ws.mybuild**, and then defined additional services in this custom build file, you would change the entry so that it looks like this:

```
* f=build/ws.mybuild
```

If a number of nodes are equipped with the same kind of network card, a single entry can be used for all of these nodes. In the example entry above, all (*****) nodes on the network would be booted from the same build file (**ws.mybuild**).

If your boot server will be servicing boot requests from two or more different nodes and/or networks, you can specify different custom build files for some of your nodes. We've provided generic build files suitable for building OS images for different network media such as Token Ring in the **/boot/build** directory.

For example, let's say you have an eight-node, NE2000-compatible Ethernet network, and that these nodes boot from the **netboot** entry **f=build/ws.ether1000**. If you added three more nodes that will boot from the same server using custom build files, you could modify your **/etc/config/netboot** so that it looks something like this:

```
9     f=build/ether2100.node9
10    f=build/ws.tr16a
11    f=build/mybuild.node11
*     f=build/ws.ether1000
```

In the example above, nodes 9, 10, and 11 use custom build files and all other nodes use the default network build file **ws.ether1000**.

Note that the position of the line with the * is important! Make sure it's the *last* line in **/etc/config/netboot**, because the file will stop being processed when the first match is found.

Setting up for broadcast booting

With a broadcasting boot ROM, **netboot** needs to know whether it should act as primary, secondary, or tertiary server with respect to each node. The utility finds out by reading an optional server mode parameter in the **netboot** file.

The general syntax of the **netboot** entries is as follows:

logical_node_ID **f=***buildfile* **|** **F=***imagefile* **[***server_mode***]**

where *server_mode* specifies the role that **netboot** is to play with respect to a node. If *server_mode* isn't specified, **netboot** will act as a primary boot server for the associated booting nodes.

The **F=***imagefile* option lets you name a custom pre-built OS image that can be downloaded to a node that boots from the network. By default, QNX builds an image on the fly using the specified build file. For more information about transmitting a named OS image to the booting node instead, see the section on "Booting a node from the network."

The possible values for *server_mode* are:

A primary server (default)

B secondary server

C tertiary server

The following example tells **netboot** to be the primary boot server for all nodes except node 4, for which it will be a secondary boot server:

```
4  f=build/ws.tr16a      B
*  f=build/ws.ether1000  A
```

For information about running **netboot** on multiple boot servers, see "Multiple boot servers."

Booting a node from a QNX network

When a node that boots from a QNX network is powered on, the code in the QNX boot ROM on the machine's network card is executed. At this point, either a specific boot server is contacted (if it's a QNX Arcnet card) or a broadcast boot request is made.

The **netboot** process on the responding boot server generates and downloads a boot image to RAM on the booting node's network card. The **sinit** program will then run and begin launching the services listed in the node's **sysinit.***node* file.

Configuring a node that boots from a QNX network involves these steps:

- inserting the QNX boot ROM
- installing the network card
- constructing the build file
- modifying the **sysinit.***node* file
- booting the node

Step 1 — Insert the boot ROM

To boot QNX from an Ethernet or Token Ring card, you must install a QNX boot ROM in the ROM socket on the card as described in the card's documentation.

Note that QNX Arcnet cards come with a special QNX boot ROM already installed. This boot ROM doesn't broadcast a boot request; instead, it contacts a specific boot server.

All other QNX boot ROMs use the broadcast-boot method. When the ROM attempts to boot, it broadcasts a boot request to a server it symbolically refers to as server "A" (primary server). If it gets no response, the ROM will start broadcasting boot requests to server "B" (secondary server), and finally to server "C" (tertiary server).

If two network cards with boot ROMs are installed in a node, the network card whose ROM resides at the *higher* memory address is the last one found during the "Power On Self Test" (POST) sequence performed by the BIOS. This card will usually be the one your node will use to send the boot request over the network.

Step 2 — Install the network card

If you haven't done so already, read the technical note that applies to the type of card you're installing (see **`/etc/readme/technotes`**).

Follow the manufacturer's installation instructions, and remember to note the physical node ID of the card(s). When booting, the QNX boot ROM will print out the physical node ID.

Step 3 — Construct the build file

You can customize the services that each node inherits through its build file. All nodes can have a distinct build file, which can be preconfigured and stored on the boot server (in the **`/boot/build`** directory) or built on the fly through the **`buildqnx`** utility.

By default, images are based on the build files specified in the boot server's **`netboot`** file and are built on the fly. When you create a

custom operating system image, an **F=***imagefile* entry must replace the standard **f=***buildfile* entry in the boot server's **netboot** file.

Net and its associated driver must be started in the build file for booting nodes. If a node will need more than two network drivers, you'll need to specify the **-d** option to **Net** in the build file. For details, see the **Net** documentation in the QNX *Utilities Reference*.

If you choose to modify a node's build file, you should make a copy of the default version and give the copy a meaningful name — such as **ws.eth1000_tr16a** for a build file that will contain both Ethernet 1000 and Token Ring network drivers — then modify the new file.

Step 4 — Modify sysinit.*node*

If you'd like to customize the services available to the node, create a **sysinit.***node* file on the boot server for the node and add the required services (see "The system initialization file" in the Basic Installation chapter).

Remember, the *node* suffix must be the logical node ID of the machine you're customizing.

Step 5 — Boot the node

The node should now boot from the network. If you have any problems, specify the **-v** option to **Net** in the booting node's build file. This will display network errors on the console. It's also a good idea to use the **-v** option for **netboot** and for the network drivers (**Net.***) as well.

Booting a node using a BOOTP server

To boot a QNX system using the BOOTP Internet boot protocol, you'll need a BOOTP ROM for your QNX client and a BOOTP server (**bootpd**) for your server. Since the TFTP protocol is used to move the image from the server to the client, you'll also need a TFTP server — this is usually provided with a BOOTP server on most systems

(UNIX, Windows 95/NT). (Our QNX 4 TCP/IP runtime package includes the **bootpd** server and the **tftpd** server.)

We support ROMs from Lanworks Technologies (**www.lanworks.com**). We've tested Lanworks ROMs with the Win32 BOOTP server from Weird Solutions (**www.weird-solutions.com**) and the QNX 4 **bootpd** server.

With the Lanworks BOOTP ROMs, the image is placed in memory at **0x10000**, so you'll need to run **buildqnx** with the **-b** option to specify the RAM address:

```
buildqnx -b 0x10000
```

For more information on setting up your QNX 4 BOOTP servers, see the **bootpd** man page in the TCP/IP for QNX *User's Guide*.

Booting a node from its own hard disk

Your network may include a node that will boot from its own hard disk. If this is the case, you must do the following before you can boot the node:

- install the network card as explained previously
- edit the node's build file
- copy the master **netmap** file to the node
- install or transfer a license to the node's hard disk

Editing the node's build file

The first entries in a build file start the Process Manager (e.g. **sys/Proc32**) and set the options that **Proc32** will assume. The **-l** ("el") option to **Proc32** assigns a logical node ID to the node.

When you installed QNX on the hard disk of this node, a logical node ID of 1 was assumed. You must change the logical node ID associated with the **-l** option to match the *node* suffix of the node's **sysinit.***node* file.

For more information about building a custom image for a node, see the "Network images" section in the chapter on building an OS image.

Copying the `netmap` file to the node

A machine that boots from its own hard disk needs a **`netmap`** file containing a logical-to-physical node mapping for *all* nodes that this machine must communicate with.

You can generate an updated "master" **`/etc/config/netmap`** file on the boot server as described in Step 7 of the "Configuring a boot server" section.

Installing or transferring a license

In order for a node to boot from its own hard disk, a license must be present on the hard disk. If you installed a new-style QNX license from the distribution media, a license for the node will already be installed in the **`/.licenses`** file. If you received a license certificate, you must add the license number to the **`/.licenses`** file. Don't forget to activate new licenses (using **`license -r`**). For more information, see the Licensing chapter.

Multiple boot servers

Since broadcasts are potentially seen by all nodes on the network, you can run **`netboot`** on several nodes, creating multiple boot servers. This lets you associate specific groups of nodes with different boot servers.

Although every **`netboot`** process usually receives each broadcast boot request, the utility responds only in accordance with its own unique **`/etc/config/netboot`** file, which defines the relationship to each requesting node (e.g. "I am node 5's secondary boot server").

No two boot servers should have the same *server_mode* entry for any given logical node ID. If this rule isn't followed, every boot server with the same *server_mode* entry for a given logical node ID will respond to that node's boot request.

Let's now look at a 15-node Ethernet network, with nodes 1 to 9 belonging to the "tech" department and nodes 10 to 15 belonging to the "marketing" department. If you wanted to make node 1 the primary server for "tech," and node 10 the primary server for "marketing," you'd run **netboot** on both nodes, each with its own **netboot** file.

On node 1 (**//1/etc/config/netboot**):

```
2  f=build/al.1000       A
3  f=build/pat.1000      A
4  f=build/celeste.1000  A
5  f=build/robin.1000    A
6  f=build/vanessa.1000  A
7  f=build/sandy.1000    A
8  f=build/john.1000     A
9  f=build/bubba.1000    A
*  f=build/ws.ether1000  B
```

On node 10 (**//10/etc/config/netboot**):

```
10  f=build/andrew.1000  A
11  f=build/bob.1000     A
12  f=build/liz.1000     A
13  f=build/fred.1000    A
14  f=build/joe.1000     A
15  f=build/betty.1000   A
*  f=build/ws.ether1000  B
```

In this case, you've also made node 10 the secondary boot server for "tech," and node 1 the secondary server for "marketing."

CAUTION: Make sure the line beginning with ***** is the *last* line in **/etc/config/netboot**, because the file will stop being processed when the first match is found.

Also, remember that the *server_mode* entry in the ***** line must specify a different role from that of all the other node entries for all the other **netboot**s running on all the other nodes on the network.

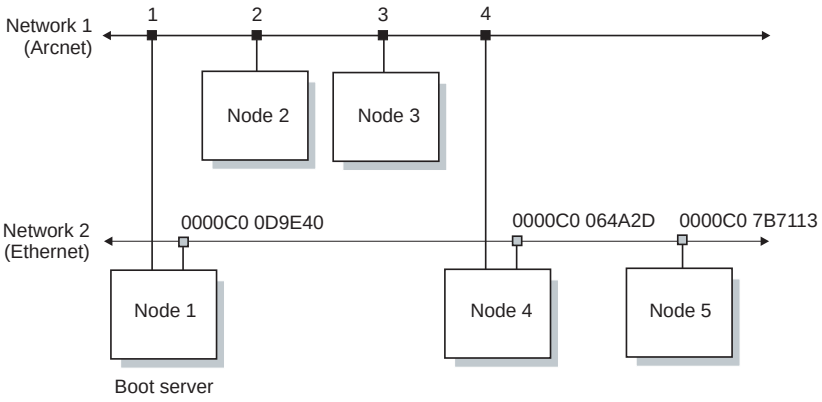
Network examples

Now that we've discussed what's involved in setting up multiple network links, let's look at how to:

- add several Ethernet nodes to an Arcnet network
- set up a fault-tolerant Ethernet network
- set up a private network link

Adding several Ethernet nodes to an Arcnet network

Let's assume the following network layout:



As you'll recall, the format of the **/etc/config/netmap** file is as follows:

logical_node_ID logical_network_ID physical_node_ID

For this example, the **netmap** file would contain the following entries:

```
1 1 t1
1 2 0000C0 0D9E40
2 1 t2
3 1 t3
4 1 t4
4 2 0000C0 064A2D
5 2 0000C0 7B7113
```

Note that the logical node IDs are unique and consistent across all interconnected networks.

In order to boot nodes on a QNX network, the **netboot** utility requires that each node have an associated OS image or build file. Note that nodes booting on the Arcnet network don't use the same build files as nodes booting on Ethernet. Consequently, the **/etc/config/netboot** file would contain the following entries:

```
2   f=build/ws.arcnet
3   f=build/ws.arcnet
4   f=build/ws.arc_ether1000
5   f=build/ws.ether1000
```

Node 1 has no entry since it boots from its own hard disk. But its **sysinit.***node* file must start both an Arcnet driver and an Ethernet driver.

You could configure node 4 to boot from either Arcnet or Ethernet.

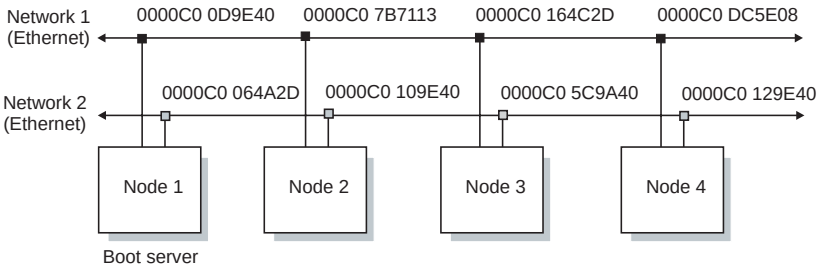
Which network node 4 boots from depends on which network card is found *last* during the ROM scan at powerup — the highest addressed card is always found last and is the one the machine will boot from.

In this example, assume we configured node 4 to boot over Arcnet by making sure the Arcnet boot ROM is at the highest address. Since node 4 is on both Arcnet and Ethernet, the node's build file starts **Net** and both the **Net.arcnet** and **Net.ether1000** drivers. This ensures the boot server can transmit the **sysinit.***node* file on both networks.

Most build files have to start the Network Manager (**Net**) and the associated network drivers. If you require three or more network drivers, you'd have to specify the **-d** option to **Net**.

Setting up a fault-tolerant Ethernet network

Let's assume the following network layout:



The corresponding **/etc/config/netmap** file would contain the following entries:

```
1 1 0000C0 0D9E40
1 2 0000C0 064A2D
2 1 0000C0 7B7113
2 2 0000C0 109E40
3 1 0000C0 164C2D
3 2 0000C0 5C9A40
4 1 0000C0 DC5E08
4 2 0000C0 129E40
```

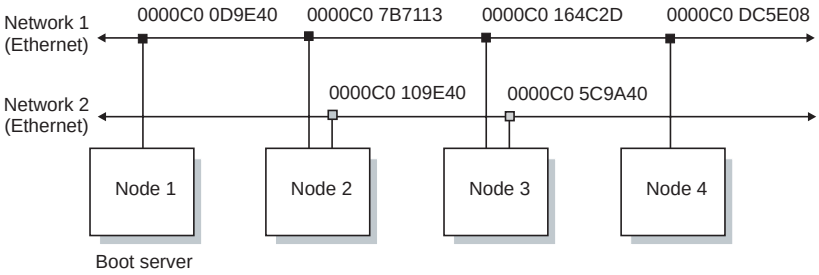
The corresponding **/etc/config/netboot** file would contain the following entry:

```
* f=build/ws.eth1000x2
```

Since an additional Ethernet driver is required, you need to create a custom build file (e.g. **ws.eth1000x2**) and add a second copy of the **Net.ether1000** driver to the file, specifying the **-l 2** option.

Setting up a private network link

If there's a lot of traffic between certain nodes on a network, you can set up those nodes with a private network link in order to offload the communication flow from the main network, as was done with nodes 2 and 3 in this example:



☞ You may need to "downgrade" the Ethernet driver's bit-transmission rate used between node 2 and node 3 on LAN 1. To do this:

➤ On node 2 and on node 3, run the driver for LAN 1 with the **`-r`** *media_rate* set to 100000 (the default is 10000000):

```
Net.ether1000 -l1 -r 100000 &
Net.ether2100 -l2 &
```

Network diagnostics

The Network Manager maintains a circular buffer of significant network runtime events. You can run the **`netinfo`** utility to display the most recent buffer. The output includes the time of the network event, the associated node number, the event's numerical code, and a brief explanation of what the event means.

For more information about these utilities, see their entries in the *Utilities Reference*.

Chapter 7

Setting up User Accounts

In this chapter. . .

Starting a user session

The **login** utility controls user access to all QNX resources and system functions when a user attempts to login to the system. A user at a terminal **/dev/ser1** can gain access to the system by entering correct responses to the following command:

```
on -t /dev/ser1 /bin/login
```

When invoked without options, the **login** utility prompts the user for a username and password. You can automate the login process through the **tinit** utility. To have **tinit** start the **login** process automatically at system startup, you would add the following entry to the node's **sysinit.***node* file:

```
tinit -t /dev/ser1 &
```

Shell initialization

The **/etc/profile** file is the first file executed by the login shell (e.g. **/bin/sh**). Local conventions can be established through this file.

The user-specific startup profile file, typically **$HOME/.profile**, is executed next. You can add any exported environment variables you need to this file. For example, if you'd like to preserve your login shell settings whenever a process starts a new shell on your behalf, be sure to define and export the **ENV** environment variable in your **.profile** file. For example:

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

This entry ensures that every time a new shell starts, the **$HOME/.kshrc** file will be executed. To display the values of all exported variables, type **export -p**.

Setting environment variables

When a user starts a session, the login shell program (e.g. **/bin/sh**) is started. See the **login** and **sh** man pages in *Utilities Reference* for info on how the shell variables and defaults operate.

Additional settings are retrieved from the user's **.profile** file, if it exists. The shell may add some of the values it requires to the user's **.profile** file automatically. For example, a user's **.profile** file may include settings for the following environment variables:

TERM defines the user's terminal type (e.g. **qnx** or **qansi** — for a list of the possible values, see the Setting up Terminals chapter)

TMPDIR gives the location of temporary workspace that utilities and programs may use (e.g. **/tmp**)

TZ sets the time zone (e.g. **TZ=EST5EDT4**)

The **set** command can set (**+**) or unset (**-**) these shell options. To display a list of the variables that have been defined for your shell, you'd type **set** at the shell prompt. To see the value of one variable, you'd type **echo $***variable_name* at the shell prompt (e.g. **echo $WORKDIR**). For more information about these shell commands, see the **sh** utility in the *Utilities Reference*.

Security

QNX provides mechanisms to control access to resources and critical system functions. These mechanisms are based on the ability of the system to identify a particular user.

When the system boots for the first time (right after the initial installation), you're automatically logged in as the superuser (**root**). This username has access to every part of the system and doesn't require a password.

To prevent unauthorized users from accessing the system, you should give **root** a password. You can create a new user account for yourself

that you can use afterwards for your daily work. The **passwd** utility allows a login password to be changed; you can use the utility to create new usernames for yourself and others.

Access control utilities

The QNX access control utilities are:

- **login** — sign on to the system
- **su** — temporarily become another user
- **passwd** — create and maintain user accounts/change passwords

The login utility

The **login** utility may be started by **tinit** on **tty** devices. The utility demands a username and password, and verifies them against the user database **/etc/passwd** before granting access to the system. If the username/password combination is incorrect, **login** displays a message to that effect and terminates. If the user enters the correct combination, **login** starts the login shell and loads that user's environment.

The su utility

The **su** utility lets you temporarily have the privileges of another user. If a user ID isn't entered on the command line when you start **su**, the utility prompts for a password that must match the superuser's password. Entering a valid access combination causes **su** to access the user database **/etc/passwd** and create a shell that has all the rights and privileges of the assumed user ID. Exiting from the shell returns you to your regular user ID.

The passwd utility

The **passwd** utility can be used to change passwords or to add a new user account to the system. Anyone can change their own password, but the superuser (**root**) is the only user who has the right to create or change other passwords. In both cases, the **passwd** utility locks

access to the user password file **/etc/passwd** to prevent any other attempts to access the file while it's being modified.

If you are:	Then use this command:
a user	**passwd** to change *your* password
root	**passwd** *username* to create or change a password

Before you start creating user IDs, you might want to modify **passwd**'s behavior. The settings in the file **/etc/default/passwd** determine which local policies will be enforced by **passwd**, such as the stringency of passwords, user ID ranges, and so on. For more information about the options you can set, see the **passwd** utility in the *Utilities Reference*.

The superuser can add a new account by invoking **passwd** with the new user's *username*. The utility prompts for information that will control the new user's environment, including access privileges. The account permissions to include are the user's group ID membership and file permissions.

For example, to create a new account with the username **jsmith**, the superuser would enter:

```
passwd jsmith
```

By default, **passwd** will then prompt for the following information:

- user ID
- group ID membership
- user's real name
- user's home directory
- initial command (e.g. **/bin/sh**)
- initial password

User ID number

By convention, user accounts have a user ID number ≥ 100. User ID numbers below 100 are often used by system processes. User ID ranges are set in the **/etc/default/passwd** file.

Group ID membership

If the group ID doesn't exist, the system will display a reminder to update the group file **/etc/group**.

The **/etc/group** file grants privileges to all users who are members of a defined group. For example, the "Finance" department would be allowed access to the company's financial records, but the "Customer Support" group would not. You can set up a group ID for "Finance" with the required access permissions, and then assign users to that group according to the privileges they require.

The **/etc/group** file contains a list of users by group. Each line has the following general syntax:

groupname`::`*group_ID*`:`*user1*, *user2*, *user3*, ..., *userN*

For example, the following entry defines the group "maestri" with group ID "123" and members "alvivaldi," "gfhandel," "gptelemann," and "jsbach":

```
maestri::123:alvivaldi, gfhandel, gptelemann, jsbach
```

Initially you'd add groups that have no users, as in the following example:

```
docs::0:
techies::1:
support::2:
marketing::3:
finance::4:
```

When a user logs in, the value in the password database **/etc/passwd** will specify the default group ID a user has been assigned to (if any). Users who belong to a group are allowed to switch to the privileges of that group any time using the **newgrp** command.

Home directory, initial command, and initial password

The home directory, initial command, and initial password items are optional. If you don't specify a value, the following defaults are used:

home directory
: **/home/***username* is used (it's created if it doesn't exist)

initial command
: the default shell startup command (**/bin/sh**) is executed immediately after the user logs in

password
: the user account is created without a password

Deleting a user account

To delete an account, you must remove *all* of the following:

- the user database entry from the **/etc/passwd** file
- the username from the corresponding group ID membership entry in the **/etc/group** file
- the corresponding encoded login password entry from **/etc/shadow** file.
- (optionally) the user's home directory
- (optionally) the user's mailbox (i.e. **/usr/spool/mail/***username*)

More about user and group IDs

Upon logging in, a user is assigned two pieces of information: a user ID and a group ID.

These are known as the user's *real* user ID and *real* group ID.

The user ID should be unique — no two users should share the same ID. This rule can be enforced by the **passwd** utility, but the superuser can override the rule by editing the password file directly.

The group ID allows several users to be associated with a group. This group mechanism lets a team of users share resources without making those resources available to the rest of the world. For more information about the **/etc/group** file and its contents, see the **newgrp** utility in the *Utilities Reference*.

Effective user and group IDs

Processes have two classes of user and group IDs: *real* and *effective*. When a process is created, it automatically inherits these four IDs from its parent (which is typically the login shell):

- real user ID
- real group ID
- effective user ID
- effective group ID

The effective user and group IDs are used for permission checking. A process can change its effective user or group ID, or both, so that they differ from their real counterparts. Typically, this is done to gain access to resources that aren't available to the real user and group IDs.

To modify effective group and user IDs:	You use the:
from within C programs	*seteuid()* and *setegid()* functions
from within the shell	**su** and **newgrp** utilities
for an executable on disk	**chgrp** and **chmod** utilities (e.g. to change the group ownership and permission bits for the **mailx** program: **chgrp mail /usr/bin/mailx** **chmod g+s /usr/bin/mailx**)

For more information on setuid and setgid bits, see the "File permissions" section.

The `newgrp` utility

The **`newgrp`** utility starts a new shell with different real and effective group IDs. When invoked without arguments, **`newgrp`** switches the group ID to the one identified in the password database for the current user. The password database controls the groups that a particular user may switch to.

File permissions

In QNX, each file has an associated set of permissions called *mode bits*. These mode bits, in conjunction with the file's *owner* and *group*, control access to the file.

When a process creates a file, the process's effective user ID and effective group ID become the file's owner and group. A set of mode bits is also assigned to the file (this is described in the "How mode bits are assigned" section).

There are three classes of mode bits:

- owner
- group
- other

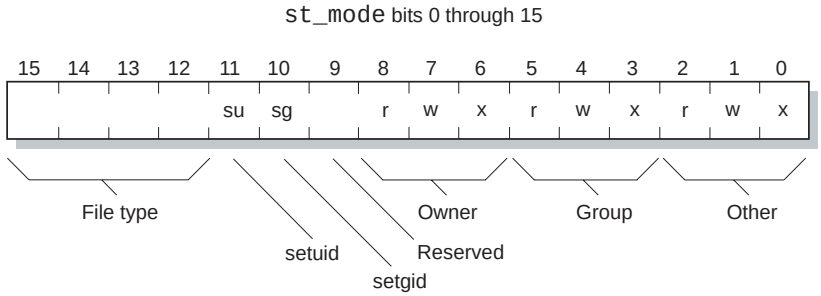
When a process tries to access a file, the Filesystem Manager honors the appropriate class of mode bits by performing a few comparisons in a specific order:

1. The effective user ID of the requesting process is compared to the owner of the file. If they match, the *owner* mode bits are honored.
2. If the owner comparison fails, the requesting process's effective *group* ID is compared to the file's group ID. If these group IDs match, the group mode bits are honored.

3 If both the owner and group comparisons fail, the *other* mode bits are honored.

As **root** (uid 0), you can access any file for read/write regardless of the permissions, and you can execute any file that has at least one mode bit permitting execute access.

The file mode bits are stored in the 16-bit **st_mode** field of the directory entry for the file; access is granted if a mode bit is set.



As the following table shows, permissions affect regular files and directories differently:

This bit:	On files, affects:	On directories, affects:
r	Reading the file	Examining the names of files in the directory (e.g. via **ls**)
w	Writing to the file	Creating, removing, and renaming files within the directory, including subdirectories

continued...

This bit:	On files, affects:	On directories, affects:
x	Executing the file	Searching the directory. This means you can change your current working directory to this directory. Also, you may include this directory within pathnames (e.g. in *open()*, *stat()*, etc.)

Execute permission has meaning only for directories and regular files; it doesn't apply to other file types.

Viewing permissions

You can use the **ls -l** ("el") command to see a file's permissions, owner, and group. The following **ls -l** output for the file **alpha.c** shows that the owner and anyone in the **techies** group may read and write to the file, while other users may only read it:

```
-rw-rw-r--  1 mplanck   techies   8475 Apr 1  1997 alpha.c
```

In the following example, the **ls -ld** output for the directory **/bin** shows that the owner and group members may read, write, and search the directory, while other users don't have write permissions and therefore can't add any new files to the directory:

```
drwxrwxr-x  2 root     techies   2048 Sep 19  1990 bin
```

But note that other users may still list or execute files in **/bin**.

It's possible — though unusual — to have read permission on a directory but not search permission. For example, **ls** could work (depending on the options you give it), but you wouldn't be able to access any file in the directory or do a **cd** to the directory.

How mode bits are assigned

The permissions on a file are derived by combining the current *umask* of the creating process with that process's requested *open()*/*creat()* mode.

The *umask* is a permission bit mask that indicates which permission bits are to be turned off if a process specifies default permission when a file, directory, or FIFO is created (see **umask** in the *Utilities Reference*).

The mask is the logical AND of the mode and the complement of the umask. For example, if the mode requests read-write permission for everyone as follows:

Binary	Octal	Symbolic
110 110 110	0666	`rw- rw- rw-`

and the umask indicates write permission for group and other:

Binary	Octal	Symbolic
000 010 010	0022	`--- -w- -w-`

then the result is read-write permission for the owner, and read permission for everyone else:

Binary	Octal	Symbolic
110 100 100	0644	`rw- r-- r--`

Changing permissions

You can change the permissions of a file with the following commands:

To:	Use:
change owner and group	**chown** and **chgrp** utilities or *chown()* function (C programs)
change access permissions; setuid and setgid mode bits	**chmod** utility or *chmod()* function (C programs)
set the file mode creation mask	**umask** utility

To modify the owner or access permissions of a file, your effective user ID must match that of the file's owner. If you change the owner of the file to be other than yourself, you'll no longer be able to change the permissions and ownership of the file. The superuser can modify the mode bits of any file, regardless of who owns the file.

setuid and setgid

To perform certain functions, a user must sometimes run a command "pretending" to be a different user or a member of a different group. Two file mode bits allow a user to do this:

- setuid (set user ID) — for example, the **passwd** utility must behave as if it were **root** so it could modify the **passwd** and **shadow** files on behalf of a user who ordinarily wouldn't have write access. (Mode bits for this command would look like this: **-rwsrwxr-x**)
- setgid (set group ID) — for example, the **mailx** command allows a user to temporarily "belong" to the group **mail**, thus allowing the user to write to other users' mailboxes, which are typically **-rw-rw----** *username* **mail**. (Mode bits for this command would look like this: **-r-xr-sr-x**)

With these bits set, an executable file will run with the privileges of its owner and group rather than with the privileges of the process that invokes the executable. These mode bits apply only to executable regular files.

When a file is loaded for execution and the setuid mode bit is set, the effective user ID of the new process becomes that of the file's owner. Similarly, if the setgid mode bit is set, the effective group ID becomes that of the file's group.

To close a potential security hole, the setuid and setgid bits are cleared any time the file ownership (owner or group) is changed. This prevents users from writing an executable file to disk, setting the setuid bit, and then changing the ownership of the file to **root**, thereby gaining unlimited system access.

The **passwd**, **login**, **su**, and **newgrp** utilities are all setuid to **root**; these programs therefore run with the permissions of the superuser.

By convention, **root** is the only user with user ID zero, which yields superuser status. With respect to access control, you must ensure that only programs that can be trusted — and absolutely need to be trusted — are setuid to **root**. No special privileges are bestowed on a program when it's setgid to **root**.

CAUTION: Since all programs setuid to **root** inherit superuser capabilities, you should make sure they do *not* have general write permissions so that only the superuser will be able to modify the programs.

The password database

These files collectively form the password database:

- **/etc/passwd**
- **/etc/group**
- **/etc/shadow**
- **/etc/.pwlock**

The access permissions to these files should be set as follows:

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--**

/etc/passwd

The **/etc/passwd** file contains a set of lines in the following format:

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

where:

username	login name of user
haspw	if empty, user has no password; if **x**, user's password is in the **/etc/shadow** file. An empty value or a value of **x** are the only permitted values.
userid	numeric user ID
group	numeric group ID
comment	a free-form comment field; must not contain "**:**"
homedir	home directory of this user (default is **/**)
shell	initial command (and arguments) to start after **login** (default is **/bin/sh**)

Here's an example **/etc/passwd** line:

```
bubba:x:290:120:Bubba L. Jones:/home/bubba:/bin/sh
```

/etc/group

The **/etc/group** file contains lines in the following format:

groupname::*group_ID*:[*user_ID*[,*user_ID*]...]

where:

groupname	name of the group
group_ID	numeric ID of the group
user_ID	one or more user IDs belonging to this group

Here's an example **/etc/group** line:

```
techies::123:bubba,charlie,sue,jake
```

/etc/shadow

The **/etc/shadow** file contains lines in the following format:

user_ID:*password*:0:0:0

where:

user_ID	name of this user
password	encrypted password of this user

Here's an example **/etc/shadow** line:

```
bubba:SRjg51|weIJ[H:819388588:0:0
```

/etc/.pwlock

The **/etc/.pwlock** file is created by **passwd** to indicate to other instances of **passwd** that the password file is currently being modified. When **passwd** finishes, the lock file is removed.

You may notice from the above permission list that **/etc/passwd** is readable by anyone. This is to provide standard utilities with a simple mechanism to find information about users. Since this file is readable, the encrypted password isn't stored in it.

The encrypted password is stored in the **/etc/shadow** file, which is readable only by the superuser. This is to inhibit unauthorized attempts to decrypt the passwords. To protect the security of your user community, you should ensure that these permissions are maintained.

QNX is shipped with a default password database that includes **/etc/passwd** and **/etc/group**. The **/etc/shadow** file isn't shipped, because the accounts initially don't have passwords associated with them.

CAUTION: If you must edit the **/etc/passwd** file on a live system (not generally recommended!), follow these precautions:

1. Run **touch** on the file to change its access time.
2. Run **ls -l** to verify that you own the file you've just touched.
3. Do your edits.
4. Remove the **/etc/.pwlock** file.

Note that if someone else (or some utility like **login**, **su**) tries to access the password database while you're editing the **/etc/passwd** file, you could end up with a corrupt password database!

Note also that if you remove a user's password, you must also remove the corresponding entry from the **/etc/shadow** file.

Default password files

The default **/etc/passwd** file that was shipped with your QNX system contains the following:

```
root::0:0::/:/bin/sh
```

The default **/etc/group** file contains the following:

```
root::0:root
```

Accounting file

The **login** utility updates system accounting information, which is logged to the **/etc/acclog** file. If this file doesn't exist, all accounting information will be discarded — this is the normal mode of operation after QNX has been installed.

For most realtime systems, this default of not keeping accounting information is recommended. If you have a dial-up line to a computer or if you run QNX on a network of many users, you may wish to change this default by creating an empty **/etc/acclog** file.

Enabling accounting

To enable accounting, you create an empty **/etc/acclog** file. You can do this using the **touch** utility:

```
touch /etc/acclog
chmod g=,o= /etc/acclog
```

Once this file is created, accounting information will be logged here.

Note that only the superuser (user ID **root**) may create and modify this file.

Record format

Each record in the **/etc/acclog** file is of the form:

ttttttttt cc data...

where *ttttttttt* is the time in seconds since 1970 (in decimal). This is always followed by a single space. The time is followed by a two-character code *cc*. This code is then followed by a space and data specific to each code. Each line is terminated by a newline character. The following utilities write to the **/etc/acclog** file:

Utility:	Purpose:	Record:
login	user logged in	*ttttttttt* **LO** *device uid gid uname*
login	login failed	*ttttttttt* **LF** *device uname*
modem	modem connect	*ttttttttt* **MO** *device baud*
su	switch user	*ttttttttt* **SU** *device uid gid uname*
tinit	start a command	*ttttttttt* **TS** *device command*
tinit	arm a device	*ttttttttt* **TA** *device*

A typical accounting file might look like this:

```
670464500 TS //1/dev/ser1 modem -b 19200 -L
670464545 MO //1/dev/ser1 2400
670464550 LO //1/dev/ser1 100 101 steve
670465824 TS //1/dev/ser1 modem -b 19200 -L
```

This record shows that **tinit** started a **modem** program to wait for calls. A call was received and answered at 2400 baud, and user ID **steve** logged in. Note that the log doesn't show a logout. The logout is inferred, because in the final entry **tinit** starts another **modem** program.

The total connect time for the user (from successful login) can be calculated like this:

670465824 - 670464550 = 1274 seconds

On a busy system, records from many devices will be interspersed throughout the accounting file. In order to match events keyed to each device, you'll find an associated node number that lets you track accounting records for all devices throughout a network in a single logfile.

Here are several common event sequences:

Event sequence:	Meaning:
TS → **LO** → **TS**	A login and logout on a dedicated line.
TS → **MO** → **LO** → **TS**	A login and logout on a dial-up line.
TA → **TS** → **LO** → **TA**	A login and logout on a dedicated line armed by a keystroke.
TS → **TS**	An unsuccessful login on a dedicated line.
TS → **MO** → **TS**	An unsuccessful login on a dial-up line.

Clearing the logfile

Once you create **/etc/acclog**, it will start to grow as records are appended to it. If left unmanaged, this file may grow to consume considerable disk space, so you should print or archive the information in this file on a regular basis. You may even want to automate this housekeeping job using the **cron** utility (see the *Utilities Reference*).

The following commands will move the accounting file to a file named by year and month, and then create a new empty log:

```
mv /etc/acclog /etc/acclogs/9607
touch /etc/acclog
chmod g=,o= /etc/acclog
```

Compressing the log file

Since the data in this file is very regular, you may use the **gzip** compression utility, which will achieve very high rates of compression on the file. This can significantly reduce disk space requirements if you keep the saved logs online or save them to a floppy.

CAUTION: Remember to move the file *before* compressing it. Never compress **/etc/acclog** directly.

Here's an example of the recommended compression procedure:

```
mv /etc/acclog /etc/logs/9607
touch /etc/acclog
chmod g=,o= /etc/acclog
gzip /etc/logs/9607
```

Note also that other utilities (possibly third-party) may add their own accounting records to the **/etc/acclog** file.

Other log files

To have QNX start recording events in a log file, simply create an empty log file in the appropriate directory. You must use the filenames given below. QNX will start logging system events as soon as the associated log file exists.

/tmp/syslog

QNX utilities log problems or unexpected events to the **/tmp/syslog** file.

/usr/adm/syslog

The **logger** utility appends lines to the **/usr/adm/syslog** file. As long as a program has write access to this file, it can run **logger** as a setuid process owned by **adm:adm**.

/usr/adm/lastlog

The **login** utility writes information to the **/usr/adm/lastlog** file (provided it exists). It can be used to track where and when a user last logged in. The **finger** utility can display information from the **lastlog** file by user.

$HOME/.lastlogin

The file **$HOME/.lastlogin** is created by **login**. It keeps track of the last time the user logged in, and from which device. The contents of this file are displayed when a user logs in. If a user hasn't logged in since the message of the day (**motd**) was last updated, the contents of the file **/etc/motd** (provided it exists) are printed to the user's terminal.

/etc/nologin

The presence of an **/etc/nologin** file prevents anyone from logging in.

/etc/wtmp

The **/etc/wtmp** file maintains a list of who is currently on the system and how these users are logged in. TCP/IP for QNX uses this information.

/etc/utmp

The **/etc/utmp** file is updated by **login**. It contains login information in binary format and maintains a history of all changes to **/etc/wtmp**.

Chapter 8

Setting up Terminals

In this chapter. . .

Terminal capabilities

Two kinds of terminal capabilities files give programmers the freedom to write terminal-independent programs. Older full-screen programs rely on the control codes in the terminal capabilities file **/etc/termcap** to make use of a terminal's capabilities. Newer programs and utilities query the **terminfo** capabilities database (under **/usr/lib/terminfo**).

Setting the terminal type

The **TERM** environment variable specifies a terminal type. Initially, a program queries the **TERM** environment variable to determine what kind of terminal it's running on. Given a terminal type, the program would then look up the terminal's capabilities either in the **termcap** file or in the applicable **terminfo** file.

The terminal type setting is typically inherited from the environment through **sinit** immediately after boot:

```
sinit -r //$(bnode)/ TERM=qnx
```

As an alternative, the terminal type can be defined in a user's **.profile** file.

For hardwired terminals that run canned applications, you can preset the **TERM** environment variable when the application is launched:

TERM=vt100 on -t /dev/ser1 *custom_application*

But you can't do that for users logging in via **login** through **tinit**. If the type of terminal is known, **tinit** can be told to define the **TERM** environment variable before launching **login**:

```
tinit -c login -t /dev/ser1 TERM=ansi &
```

A word about dial-up access

Unless you restrict dial-up access to only a particular type of terminal, modem users will need to run the **termdef** utility to query for a terminal type. The **termdef** utility can display all of the supported terminal types and ask the user to select one:

```
tinit -c "modem -c termdef" -t /dev/ser1 &
```

Once a terminal type has been selected, **termdef** sets the **TERM** environment variable accordingly and presents the **login** prompt to the user.

The termcap database

Control codes in the QNX **termcap** file can define the input and output capabilities of a number of terminal types, including:

- Generic ANSI standard terminal (**ansi**)
- QNX console (**qnx**)
- QNX ANSI (**qansi**, **qansi-g**, **qansi-t**, **qansi-m**, or **qansi-w**)
- QNX terminal (**qnxt**)
- QNX terminal with mouse events (**qnxm**)
- QNX windows (**qnxw**)
- QNX monochrome terminal or console (**qnxtmono**)
- QNX 2.15 serial terminal (**qnxt2** or **qnxs2**)
- VT100 with Auto-margin (**vt100**)
- VT102 (**vt102** or **vt102-plus**)
- X-term terminal emulator (**xterm**, **xterms**, **xterm-m**, or **xterm-q**)

For a complete list of terminal types, see **/etc/termcap**.

Each entry in the **termcap** file begins with a line of terminal aliases. The remaining sequence of control codes is a list of terminal capabilities.

Each capability string can be identified by a two-character name. In general, a string describes the terminal's behavior when the user presses a key, or the visual effect displayed through program control.

If there's more than one entry for the same terminal type, the first entry is used.

The terminfo database

The **terminfo** subdirectories **/usr/lib/terminfo/...** contain binary files. Each file in the **terminfo** database contains the capability definitions associated with a single terminal type. QNX is shipped with several of these files, including:

/usr/lib/terminfo/q/qnx (the QNX console)

/usr/lib/terminfo/v/vt100 (a VT100 terminal)

/usr/lib/terminfo/a/ansi (an ANSI terminal)

The corresponding source definitions for all QNX-supported terminal types are contained in **/usr/lib/terminfo/terminfo.src**.

The **termcap** and **terminfo** source files are similar in that they both contain descriptions of terminal capabilities. However, the capability names in a **terminfo** file can be two to five characters long. And because the **terminfo** files are compiled, they load and execute faster.

If you need to convert from **terminfo** to **termcap** (as is the case for some older applications), the **infocmp** utility has a **-C** option that generates a **termcap** entry from an existing **terminfo** definition. If there's more than one entry for the same terminal type in a **terminfo** file, the last entry is used.

If you don't have a **terminfo** file for your type of terminal, you could look for the appropriate **terminfo** file on any UNIX system, or create a new **terminfo** file through the **infocmp** and **tic** utilities.

Creating a terminfo file

To view or change the information in an existing **terminfo** capabilities file, you must first decompile the file through the **infocmp** utility. To use a modified source file, you must recompile it into binary form using **tic**, the **terminfo** compiler utility.

Applications can extract the capabilities they need for a given terminal directly from the compiled version.

You may want to choose capabilities from the higher-level library **curses** instead.

Extended capabilities for mouse events

The QNX **termcap** and **terminfo** files support many of the standard terminal capabilities. In addition to these, QNX supports the following extensions to the string and numeric capabilities to handle mouse events.

termcap name	**terminfo name**	**Description:**
ZZ	*mcud1*	micro_down. Don't return press.
Za	*mcub1*	micro_left. Return mouse movement.
Zb	*mcuf1*	micro_right. Don't return mouse movement.
Zd	*mcuu1*	micro_up. Don't report release.
Zf	*mcud*	parm_down_micro. Return ADJUST press.
Zg	*mcub*	parm_left_micro. Return SELECT press.
Zh	*mcuf*	parm_right_micro. Return MENU press.
Zi	*mcuu*	parm_up_micro. Report button release.

continued...

termcap name	**terminfo name**	**Description:**
ZJ	*smicm*	enter_micro_mode. Report screen size changes.
ZT	*rmicm*	exit_micro_mode. Don't report screen size changes.
Yd	*maddr*	max_micro_address. Maximum value in micro_..._ address (a numeric capability).

For more information

For a complete list of capability names and their meanings, see the **terminfo** technote on QUICS. For more information about **termcap** and **terminfo**, we recommend *termcap & terminfo*, by Strang, Mui, & O'Reilly (ISBN 0-937175-22-6).

Chapter 9

Print Spooling

In this chapter. . .

Introduction

Sharing resources on a network

QNX encourages the network-wide distribution of resources with few artificial boundaries. Every device (disk, modem, printer, etc.) connected to a computer is, by default, a *shared* resource — a program running on any computer has equal access to any device on the network, whether the device is connected to the same computer (local) or to another machine (remote).

Although it's easy for a user to transparently access any resource on the network, the system administrator may need to take steps to control access to certain types of resources. Most printers, for example, can be used by only one user at a time and therefore require some sort of enqueuing facility to avoid conflicts. To allow convenient access to printers, QNX provides a set of *spooling* services.

Spoolers

A *spooler* is simply a mechanism that accepts requests for a resource and then allocates the use of the resource according to a set of specified rules.

To understand how a spooler can be useful, let's look at how it controls access to a printer. A printer must be available at all times to users, yet it can print only one job at a time. If a spooler is present, users can send their data through the spooler rather than directly to the printer. Upon receiving data destined for the printer, the spooler writes this data into a temporary file instead of sending it immediately to the printer. Later, when the printer becomes available, the spooler will write the data to the printer. Thus, many users can freely submit print jobs to one physical printer.

QNX implements spooling through the use of named queues that are referenced by the "**`lp`**" set of utilities; these queues also reside in the file namespace in the **`/dev/spool`** directory. Data written to a queue will be placed on an internal list and ultimately sent to a defined output device.

The QNX spooling server (**lpsrvr**) can maintain many different spool queues. The following utilities operate on spool queues:

lp	submit files to a spool queue
lprm	remove jobs from a spool queue
lpc	control spooler queue
lpq	display spool queue status

For more information on these utilities, see the *Utilities Reference*. For information about how to queue print jobs to a printer on a TCP/IP network, see the chapter on remote printing in the TCP/IP for QNX *User's Guide*.

Using the spool utilities

Starting the spooler

Before any spooling can occur in a QNX system, you must run **lpsrvr**:

```
lpsrvr &
```

To determine what resources it has available, and how it's expected to manage them, the **lpsrvr** utility first looks for a setup file called **/etc/config/lpsrvr.***node* (where *node* is the node ID of the node **lpsrvr** is running on). If no setup file is found with a **.***node* extension, **lpsrvr** will use the **/etc/config/lpsrvr** file.

Submitting spool jobs

The following **lp** command will cause the file **report** to be inserted into the default spool queue and ultimately printed:

```
lp report
```

For more information on the default queue, see "The spool setup file" in this chapter.

In systems where more than one spool queue is available, you can specify the queue name. The following command inserts **report** into a spool queue called **txt**:

```
lp -P txt report
```

You could also use a command that writes directly to the queue file:

```
cp report /dev/spool/txt
```

Querying spool jobs

To examine the spool queue, you can use the **lpq** utility. The following is a sample output from **lpq**:

```
1:  fred   [job #39]   1400 bytes  lalist.doc
2:  wilma  [job #42]   2312 bytes  netdrvr.c
```

This utility lets you determine when any submitted jobs have been completed; it also provides the spool job ID for use with other **lp** utilities.

Canceling spool jobs

The **lprm** utility lets you remove jobs from a spool queue. You can remove a job explicitly by specifying its job ID number. Given the state of the default queue shown above, **fred**'s job (#39) could be canceled with this command:

```
lprm 39
```

If job #39 is currently in progress, it will be abandoned. The success of abandoning current spool jobs may vary with the type of output device you're using — some printers have large internal buffers.

The superuser may also remove all jobs belonging to a particular user. For example, all of **fred**'s jobs can be canceled with the following command:

```
lprm fred
```

Controlling the spool queues

The **lpc** utility is a system administration tool for managing spoolers. It lets you perform many control functions, such as starting up or shutting down a queue. The following basic functions are provided:

- suspend/resume enqueuing of jobs
- suspend/resume dequeuing of jobs
- suspend/resume the current job
- delete the current job
- rearrange the jobs in a queue
- move jobs to a different queue
- display the status of queues

Note that **lpc**'s functionality overlaps that of **lpq** and **lprm**. This overlap is convenient, because unlike **lpq** and **lprm**, **lpc** can be used interactively.

Spooler architecture

The QNX spooling system is based on two objects: *queues* and *targets*. These work with each other to provide a flexible method of controlling data transformations and queuing.

A queue is an internal list of pending data to be sent to a target. As mentioned earlier, each queue is given a name that users specify when submitting jobs.

A target is associated with the physical output device (e.g. a printer) and removes jobs from queues. You can connect the output of a queue to one or more targets or connect multiple queues to a single target. However, you can connect the output of a target to only one device.

Queues can have optional attributes called *filters*, which are of two types:

- copy-in (**ci**) filters, which perform operations on the data before it's copied onto a queue
- copy-out (**co**) filters, which operate on the data after it's removed from a queue.

For example:

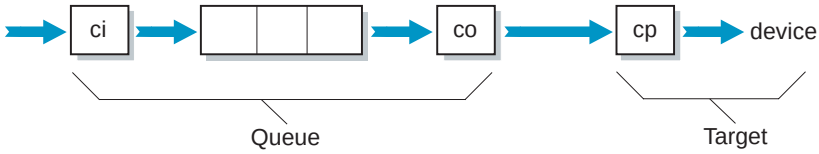
```
ci=a2ps -H"$(file)" | awk '/%%EndProlog/ { print "<< /Duplex \
    true >> setpagedevice"; } { print $0; }'
co=echo $(username)"\n" "put" $(spfile) | SOCK=666 /usr/ucb/ftp net_printer
```

Targets may have an optional *control program* (**cp**) that allows the output device to be initialized between jobs if required. For example, if a job were canceled and the device needed to be primed again, a device-specific control program could detect SIGTERM and take whatever action necessary to restore the device to a stable state.

The following diagrams illustrate how queues, filters, and targets can be configured to work with each other.

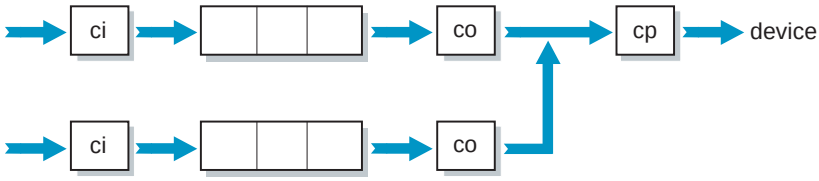
One queue feeding a single target

The following configuration could be used where there's a single printer and where a single (or no) translation of the data is required:



Multiple queues feeding a single target

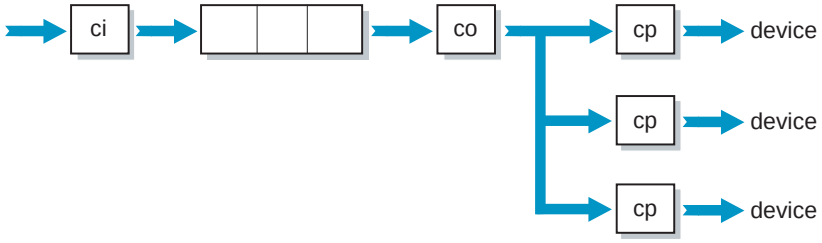
Multiple queues may feed a single target, in which case the target will select the appropriate job from all the jobs in those queues based upon queue priority, then upon time of submission (i.e. the oldest pending job).



At the end of this chapter, you'll find several example setup files. The above configuration is used in the example file in which one queue converts ASCII to PostScript, while the other is a direct PostScript queue. Both queues feed the same target, which sends the data to a PostScript printer.

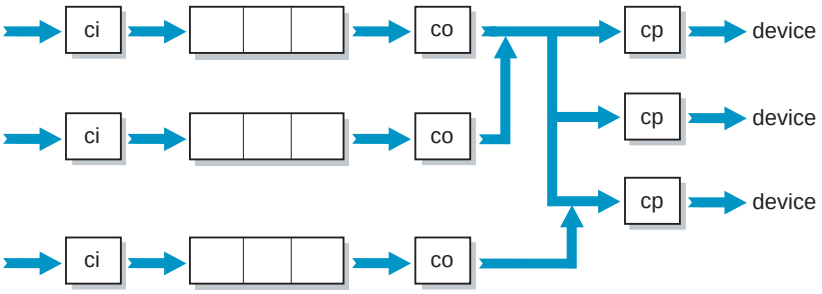
One queue feeding multiple targets

If the output of a queue is sent to many targets, the spooler will select whichever target is available. The following configuration is useful if you have three printers side by side and it doesn't matter which one prints your job.



Multiple queues feeding multiple targets

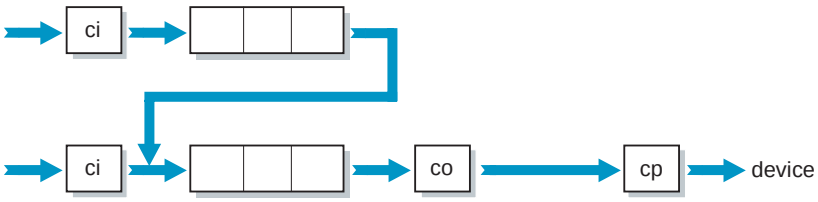
The following configuration is a combination of the previous two examples. A third queue has a separate channel to one of the targets. This channel could be used to ensure that jobs requiring the third printer are always sent to that printer (e.g. the printer has color capabilities).



Chaining queues

The output of a queue is usually sent to one or more targets, but it's possible instead to *chain* the output into another queue.

With chaining, the output of a queue is placed directly onto the destination queue, thus avoiding any possible copy-in operation on that queue. If the final queue in a chain has a copy-out filter, the filter will be applied.



The spool setup file

When started, the spooler accesses a file to get its configuration information. If no file is specified on the command line, the spooler uses the **`/etc/config/lpsrvr`** file. This file defines queues, targets, and the relationships between them.

Syntax definitions

Queues and targets have symbolic names as well as a set of attributes. Each entry in the setup file has the following format:

[*name***]**
attribute
attribute
.
.
.

The definition of a queue or target begins with a **[***name***]** directive and consists of all valid attribute specifications until the next **[***name***]** directive. All leading white space is ignored. Comment lines start with a pound sign (**#**).

To continue a single attribute beyond one line, you must put a backlash (\) right before the newline character.

The name may be up to 48 characters long and may contain only alphanumeric characters.

If the object being described is a target, the name must be preceded by a dash (**-**). The dash is for delineation only; it isn't considered part of the name.

Each attribute consists of a two-letter key in one of the following forms:

- *key* (Boolean)
- *key***#***number* (numeric)
- *key***=***string* (character)

All *numbers* are assumed to be decimal numbers, unless they start with a leading zero (meaning octal) or a leading 0x (meaning hex).

All *strings* contain printable characters. The backslash (\) is a "special" character. It can be used to escape other characters. For example, a "real" backslash must be represented with: \\

The following table describes all defined keys, including the default used for each key if its corresponding attribute isn't specified. In the **Use** column, "Q" means "queue," "T" means "target," and "G" means "global."

Key:	Description:	Default:	Use:
ab=*string*	Executed when target *ab*andons command for any reason	nil; no command is executed	T
af=*string*	*a*ccounting *f*ile that commands may use	accounting isn't performed (**lpsrvr** logging is unaffected)	Q

continued...

Key:	Description:	Default:	Use:
cd=*string*	directory where temporary spool files are found	**/usr/spool/lp**	G
ci=*string*	*c*opy-*i*n command; used before placing the job on the queue	binary copy-in of job; no transformation applied	Q
co=*string*	*c*opy-*o*ut command; used when removing job from queue	binary copy-out of job; no transformation applied	Q
cp=*string*	device *c*ontrol *p*rogram; used by the target	binary copy of job; no transformation applied	T
dv=*string*	*d*evice that the target uses	output is sent to standard output of **lpsrvr**	Q, T
mn#*numeric*	*min*imum number of jobs for queue before flushing	**0**; jobs are despooled ASAP	Q
mx#*numeric*	*max*imum number of jobs the queue will hold	no limit; queue limits are based on memory and disk space	Q
na=*string*	*na*me; a string that describes the queue or target	nil string	Q, T
ok=*string*	command executed when target completes normally	nil; no action is performed	T
pr#*numeric*	*pr*iority to run this queue at (1-100); 100 is the highest	50	Q
qn=*string*	*q*ueue to chai*n* a despooled job onto	nil; delete job after despooling	Q
re#*numeric*	*re*tries when a job fails	0	Q
rt#*numeric*	*r*etry *t*ime; number of seconds or die	forever	Q

continued...

Key:	Description:	Default:	Use:
sp=*string*	registered name of *sp*ooler maintaining this queue	**/qnx/spooler**; registered only if no queue specifies a name	G
ta=*string*	*ta*rget to be associated with this queue	output is sent to standard output of **lpsrvr**	Q
wa#*numeric*	*wa*it this number of seconds before despooling each job	**0**; jobs are despooled ASAP	Q

Since the keys are case-sensitive, we reserve all keys formed by two lowercase letters. You can safely implement custom extensions by using uppercase or mixed-case keys. The spooler utilities will ignore any options they don't understand.

Global keywords

Two keywords, **sp** and **cd**, can define global information for the spooler. To make them apply globally, you precede these keywords with a pair of empty brackets (**[ ]**).

Registering names — sp

Since the spooler always adopts the **/dev/spool** file namespace, only one spooler may run on each node. You can run multiple spoolers on your network if each spooler registers a different global name. Otherwise, each spooler will attempt to register the same default global name **/qnx/spooler**. With multiple spoolers, you may wish to use names such as **/qnx/spooler2**, **/qnx/spooler3**, and so on.

To specify the global name to be registered by a spooler, you use the **sp** command in the setup file. The name must always begin with a leading slash (**/**). If no **sp** keyword is specified, the default name **/qnx/spooler** is globally registered (i.e. network-wide).

Specifying a temporary directory for spool files — `cd`

The **`cd`** keyword lets you define the directory to use for the creation of the spooler's temporary spool files. By default, temporary files are created in the **`/usr/spool/lp`** directory. These files get deleted when they're no longer required.

Any path you specify to the **`cd`** keyword should begin with a leading slash. Note that the specified directory must exist, with appropriate access rights assigned.

The following example setup file informs the spooler to register the name **`/qnx/spooler2`**; it also places temporary files in the **`/tmp/spool2`** directory:

```
# Global spooler variables

[~]
    sp=/qnx/spooler2
    cd=/tmp/spool2

# Text queue
[txt]....
```

Variables

The spooler will set the following variables appropriately when it encounters them:

Variable:	Description:
$*(file)*	the filename of the submitted file or "**`--standard input--`**" if no name is provided
$*(fname)*	the fully resolved pathname of the submitted file or "**`--standard input--`**" if no name is provided
$*(spfile)*	the name of the spool data file

continued...

Variable:	Description:
$*(username)*	the login name of the user who submitted the job
$*(userid)*	the numeric user ID of the user who submitted the job
$*(queue)*	the name of the queue the job currently belongs to
$*(target)*	the name of the target the job currently belongs to
$*(device)*	the name of the device the job is scheduled on
$*(ncopies)*	the number of copies the user requested
$*(jobid)*	the job ID number of this job
$*(cifile)*	a temporary name that copy-in routines may use

In addition, all the keys defined above can be referenced as variables. For example, $*(ci)* will expand to the name of the copy-in command, **ci=***string*.

Default behavior

The **cat** command is used as the default copy-in and copy-out filter commands. Also, unless otherwise specified in the setup file, the standard input and standard output of filter commands are automatically connected to default files or processes. The default for copy-in is as follows:

```
cat < $(fname) > $(spfile)
```

There are two possible defaults for a copy-out, depending on whether or not a *control program* (i.e. **cp**) is defined:

```
cat < $(spfile) > $(device)
cat < $(spfile) | $(cp) > $(device)
```

For example, a setup file like this:

```
[txt]
```

```
    ta=lpt
    ci=pr -f -h
    co=txt2ps
    cp=init_printer

[-lpt]
    dv=/dev/par
```

would result in the following substitutions:

```
pr -f -h < $(fname) > $(spfile)
txt2ps < $(spfile) | init_printer > /dev/par
```

Using setup files

Queues and targets

When you create a setup file, you specify each queue by giving it a name and an optional list of parameters; you specify each target by starting its name with a dash (**-**). To illustrate, here's a very simple setup file:

```
[txt]
    ta=lpt

[-lpt]
    dv=/dev/par
```

In this file we have a queue called **txt** and a target called **lpt**. When data is sent to the **txt** spool queue, the data is saved in a temporary spool file and is known as a "job." When the spooler removes that job from the queue, the job is placed on the target, **lpt**, which then directs the data to the parallel port, **/dev/par**.

Filters

It's often necessary to filter spooled data, so **lpsrvr** provides the *copy-in* and *copy-out* mechanisms. As mentioned earlier, copy-in is run before the data is placed on the queue, while copy-out is run after the data is removed from the queue.

Note that if you have many queues feeding a single target and one of the queues has a copy-out filter that may take a long time to run, the target will be temporarily unavailable to the remaining queues if that queue is selected. (For more information, see the "Example setup files" section.)

It's good practice to put double quotes around variables to ensure they're "safe" for the command.

Using copy-in

A good example of the use of a copy-in filter is to pass the data through **pr** to paginate it:

```
[txt]
    ci=pr -f -h "$(file)"
    ta=lpt

[-lpt]
    dv=/dev/par
```

Using copy-out

You might use a copy-out filter, for example, when you have both a PostScript printer and an HP printer. You could have two copy-out filters to generate the proper output format. Let's say you have two programs: **txt2ps**, which generates PostScript, and **txt2hpgl**, which generates HPGL:

```
[ps]
    ta=lpt1
    co=txt2ps
```

```
[hp]
    ta=lpt2
    co=txt2hpgl

[-lpt1]
    dv=/dev/ser

[-lpt2]
    dv=/dev/par
```

Don't hardcode the output device in a copy-out definition; use targets for this purpose. Targets must have *exclusive* use of a device and should be the only means of accessing a device within the spooling system.

Chaining queues

Queues can be chained — this moves the job from queue to queue. The following is a simple example of chaining onto a queue named **tmp**, which then sends the data to the **lpt** target:

```
[txt]
    qn=tmp

[tmp]
    ta=lpt

[-lpt]
    dv=/dev/par
```

Accounting information

The **af** keyword lets you specify a filename that the commands may access as **$(af)** so they can write any information they want to log.

Input errors

The invoking **lp** utility will report any errors that occur during the input of data into a queue. If you submit data by directly writing to the spool file in the **/dev/spool** directory, write errors will occur if something prevents a successful copy into the queue.

Output errors

You can use the **ab** keyword to specify a command to execute if a target abandons a job. The following would inform the invoking user of the error via a mail message:

```
ab=echo lpsrvr print job $(jobid), file $(file) failed \
| mailx $(username)
```

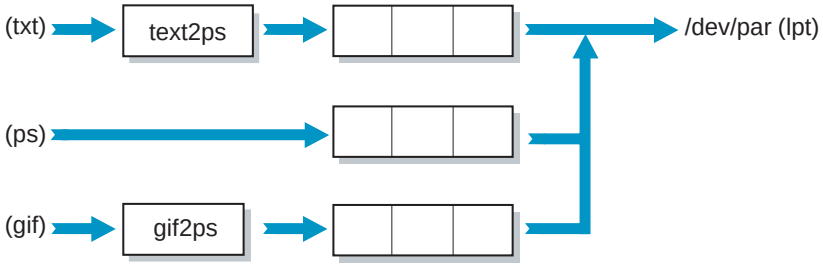
Example setup files

Multiple queues feeding a single target

The following example shows a set of queues that share a common target. The three queues are named:

txt (ASCII text files)
ps (PostScript files)
gif (Graphic Interchange Format files)

The configuration is as follows:



Here's the file to set this up:

```
# ASCII to PostScript queue:
[txt]
    ta=lpt
    ci=text2ps
    pr#50

# direct PostScript queue:
[ps]
    ta=lpt
    pr#60

# GIF to PostScript queue:
[gif]
    ta=lpt
    ci=gif2ps
    pr#5

# target printer:
[-lpt]
    dv=/dev/par
```

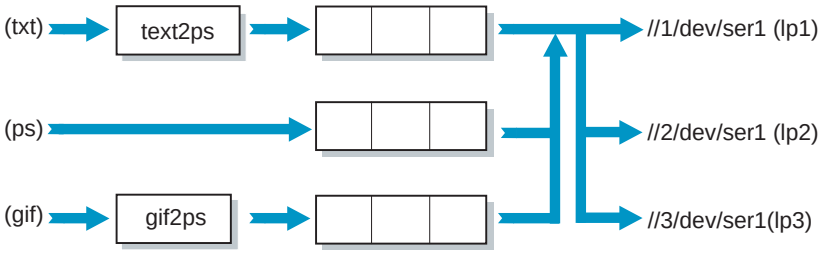
In this example, users send files with the **lp** utility to the appropriate queue, which converts the file through a copy-in filter (except for the **ps** queue). The queue then sends the converted data to the target, **/dev/par**.

Since the hypothetical filter programs **text2ps** and **gif2ps** may take a relatively long time to process a job, copy-in filters are used rather than copy-out filters. A copy-out filter will tie up the target, preventing other jobs from being dequeued while the filter is running. The chosen configuration allows other jobs to be sent to the target while the PostScript translation is being generated.

Also, since GIF files tend to be large, they're assigned a lower priority than the others.

Multiple queues feeding three targets

The following example is a further refinement of the above setup, with some additional features required for a larger configuration. There are now three printers, all PostScript, located in different parts of the building (connected to **//1/dev/ser1**, **//2/dev/ser1**, and **//3/dev/ser1**). The configuration looks like this:



Here's the file to set this up:

```
# ASCII to PostScript queue:
[txt]
    ta=lp1,lp2,lp3
    ci=text2ps
    pr#50

# direct PostScript queue:
[ps]
    ta=lp1,lp2,lp3
    pr#60

# GIF to PostScript queue:
[gif]
    ta=lp1,lp2,lp3
    ci=gif2ps
    pr#5

# target printers:
[-lp1]
    dv=//1/dev/ser1
    ok=echo file $(fname) sent to $(target) \
    | mailx $(username)
    ab=echo file $(fname) did not get printed \
    | mailx $(username)

[-lp2]
    dv=//2/dev/ser1
    ok=echo file $(fname) sent to $(target) \
    | mailx $(username)
    ab=echo file $(fname) did not get printed \
    | mailx $(username)

[-lp3]
    dv=//3/dev/ser1
    ok=echo file $(fname) sent to $(target) \
    | mailx $(username)
    ab=echo file $(fname) did not get printed \
    | mailx $(username)
```

The above configuration uses the same three queues described earlier (**txt**, **ps**, and **gif**) but they now feed three separate targets. The spooler selects the first available target from the set of targets (**lp1**, **lp2**, **lp3**) and sends the data to its corresponding printer.

Queue selection is based on priority first, then on the age of the job.

In this example, a mail message is sent to the submitter indicating whether or not the job completed normally.

Accessing spoolers and queues

Many spoolers may be present on a network; each of these can maintain multiple queues. To communicate with any given spooler, the **lp** utilities first locate the spooler through the unique global name that the spooler registers (see the "Global keywords" section).

When using any **lp** utility, you can:

- omit both the spooler and the queue
- specify only the spooler
- specify only the queue
- specify both the spooler and the queue

If you don't specify the spooler on the command line, the utility checks the **LPSRVR** environment variable, which contains the name of the default spooler. However, if **LPSRVR** isn't defined, the utility will use the spooler that has registered the default global name **/qnx/spooler**.

If the utility successfully locates a spooler, but you haven't specified a queue on the command line, the utility will check the **LPDEST** environment variable, which contains the name of the default queue. However, if **LPDEST** isn't defined, the utility will use the first queue entry in the setup file of the located spooler.

LPSRVR and LPDEST

The **LPSRVR** and **LPDEST** environment variables are used when the command-line information given to the **lp** utilities doesn't fully specify which spooler or queue to use.

LPSRVR

LPSRVR specifies the default spooler. The following setting of **LPSRVR** would indicate that the default spooler is the one with the name **/qnx/spooler2**:

```
export LPSRVR=/qnx/spooler2
```

The name specified in **LPSRVR** must always begin with a leading slash. **LPSRVR** must never contain the name of a queue.

LPDEST

LPDEST specifies the default queue. You can use **LPDEST** in two ways. You can specify a queue only:

```
export LPDEST=waybills
```

Or you can specify a spooler and a queue:

```
export LPDEST=/qnx/spooler2/waybills
```

If you specify a spooler in **LPDEST**, the **LPSRVR** variable is ignored, even if it's defined.

For compatibility, the **lp** utilities look at the **PRINTER** environment variable if **LPDEST** isn't present.

Examples

Let's look at a few simple examples that show some of the ways you can specify spoolers and queues. For these examples, let's assume you have two spoolers on the network, each with two queues.

The first spooler uses the default global name, **/qnx/spooler**; its queues are named **txt** and **ps**. The second spooler uses the name **/qnx/spooler2**; its queues are named **checks** and **waybills**.

First spooler:	Second spooler:
/qnx/spooler/txt	/qnx/spooler2/checks
/qnx/spooler/ps	/qnx/spooler2/waybills

Naming neither a spooler nor a queue

Let's say you enter the following **lp** command, specifying neither a spooler nor a queue:

```
lp test.dat
```

The utility will first try to locate a spooler by checking **LPSRVR**. If that variable isn't defined, the utility will locate the spooler with the default global name **/qnx/spooler**.

The utility will then try to determine the queue by checking **LPDEST**. If that variable isn't defined, the utility will select the first queue specified in the setup file of the located spooler.

Naming only the spooler

If you specify a string that begins with a leading slash, the string is always assumed to be a spooler name. Thus, if you enter the following command, the utility will treat **/qnx/spooler2** as a spooler:

```
lp -P /qnx/spooler2 test.dat
```

Because the second spooler uses **/qnx/spooler2** as its name, that spooler will be located. The utility will then try to use **LPDEST** as the default queue. If **LPDEST** isn't defined, the job is submitted to the queue **checks**, since that's the first queue in the setup file of **/qnx/spooler2**.

Naming only the queue

If you specify a string that *doesn't* begin with a leading slash, the string is always assumed to be a queue name. Thus, if you enter the following command, the utility will treat **`txt`** as a queue:

```
lp -P txt test.dat
```

The utility will first try to locate a spooler by checking **LPSRVR**. If that variable isn't defined, the utility will use the first spooler, since that spooler has registered the default global name, **`/qnx/spooler`**.

Naming both the spooler and the queue

In the following example, both the spooler and the queue are named. Since the string begins with a leading slash, the utility will initially attempt to find a spooler called **`/qnx/spooler2/waybills`**.

```
lp -P /qnx/spooler2/waybills test.dat
```

However, since the spooler **`/qnx/spooler2/waybills`** doesn't exist, the search will fail, at which point the utility will treat the specified string as a spooler name followed by a queue name. The spooler with the name **`/qnx/spooler2`** will be located, and jobs will be submitted to the **`waybills`** queue on that spooler.

Initialization files

You might find it useful to configure several default spoolers, if, for example, your marketing, sales, and R&D departments each has a spooler running:

`LPSRVR=/qnx/spooler` (marketing)
`LPSRVR=/qnx/spooler2` (sales)
`LPSRVR=/qnx/spooler3` (R&D)

You normally initialize environment variables beforehand in system initialization files. You may wish to use one of the following files to initialize the variables:

`/etc/config/sysinit.`*node* (run at boot time)

`/etc/default/login` (run at login time)
`/etc/profile` (run by every login shell).

Chapter 10

Making Backups

In this chapter. . .

Introduction

This chapter deals with making a copy of your data to guard against hardware, software, or human error that may destroy the original. If your data is important to you, you should follow a backup routine that would allow you to restore lost data with minimal cost to you in time and money.

Remember: hard disks *do* fail and people *do* make mistakes. It's too late to start a backup after your data is gone!

You can back up an entire filesystem or only portions of it. Users may elect to back up their own files only, usually to floppies. To back up large portions of the filesystem with files owned by many users, you'll need to have read permissions on these files. The superuser (**`root`**) has such privileges.

When to back up

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.

Backup formats

QNX supports a variety of backup formats that can be classified into two groups:

- archives
- regular filesystems

An *archive* consists of one or more files, merged into a single unit with its own directory of contents. This single unit can be saved either to a regular QNX file or to a raw block device like a floppy or tape.

Saving to a *regular filesystem* simply involves copying the files. In this case, the destination must be a device with a mounted QNX filesystem on it.

Archive backups

QNX supports three major archive utilities:

- **cpio**
- **tar**
- **pax**

Both **cpio** and **tar** are implemented as links to **pax**, which is capable of reading and writing both **cpio** and **tar** formats. Both **cpio** and **tar** are common on UNIX systems. The **pax** utility is a cover utility for **cpio** and **tar**, so it doesn't support its own archive format. By default, **pax** will use the **tar** format when creating an archive.

The **pax** utility will detect when you've reached the end of a disk or tape on a volume and will prompt you to insert the next volume to be used for the save. The result is a backup spanning several media (disks, tapes, etc.).

Naming volumes

Unfortunately, the **tar**/**cpio** format does *not* label the media with volume IDs. If you mixed up your media or inserted the wrong one out of order, you would end up restoring bad data.

To safeguard against this possibility, QNX is shipped with the **vol** utility, which labels each disk volume with its sequence number and therefore prevents you from inserting out-of-sequence media.

The **vol** utility can be used with floppies or removable disks. This utility isn't for use with tape systems.

The **vol** utility will, by default, step over block one of all media. This is important for floppy diskettes and cartridge disks that contain a QNX signature in block one. The signature contains the size of the

diskette (360K, 1.2M, 1.4M, etc.) and allows for automatic remounting of removable media by the filesystem.

We ship QNX distribution diskettes using **pax** to create an archive, **freeze** to compress the data, and **vol** to write to floppies.

If you wish to save data for restoration on a UNIX system, don't use **freeze** or **vol**: you won't find those utilities to do the restore at that end. Instead, use **pax** to save and restore directly to the target media.

Filesystem backups

You back up to a filesystem by copying files, probably with the **cp** utility or with **cpio -p**. If your destination is a floppy, the **cp** utility will prompt you for more diskettes, but remember that no single file can be larger than the diskette's capacity. If you wish to back up to floppy, we recommended that you use one of the archive utilities.

Backup media

Your choice of backup media will be determined by available hardware and cost. Here are some common choices:

- floppy
- tape
- removable disk
- fixed disk

QNX provides drivers for several SCSI controllers. These drivers scan for hard disks, sequential-access tape units, WORM (write once/read multiple) devices, CD-ROM drives, and optical drives.

Floppy

Floppies are the most widely available device for personal backups. Their major shortcoming is their limited size. Since the QNX **pax** and **vol** utilities let you span media, your only concern will be having to

feed several floppy diskettes into the drive. If you have to deal with more than four or five floppies, this will make the procedure unpleasant enough that you may stop doing it regularly! You might want to consider compressing your data as described below.

In order to back up/restore from a floppy diskette, you must make sure the floppy driver (**Fsys.floppy**) has been started (see **Fsys.floppy** in the *Utilities Reference*.

The following command will start the floppy drive (assuming that **Fsys** is already running):

```
Fsys.floppy &
```

By default, **Fsys.floppy** creates a block special device for floppy drive 0 (**/dev/fd0**). If your machine has more than one floppy drive, **Fsys.floppy** creates a block special device for each. For example, drive 1 would map to **/dev/fd1**.

Archiving data to floppy

When you use an archive utility for your backups, it reads and writes directly to the first block. Because **tar** writes archive data directly to the first block, the first block on the diskette will contain 512 bytes of raw data. This could pose a problem for **Fsys.floppy**, which attempts to read the first block on all floppies for the QNX signature.

To prevent unpredictable behavior when archiving data or restoring archived data from floppy diskette, run the **lockfd** utility before you start. The **lockfd** utility causes **Fsys.floppy** to lock the driver to a specific media size and suppresses the driver's attempt to read signature information from the first block if the data has to be restored from floppy.

Copying files to floppy

You must format and initialize any unformatted diskettes before you can mount the floppy device and copy files to the floppy. The following example would initialize a high density $3\frac{1}{2}''$ floppy diskette:

```
fdformat -s 1.4m /dev/fd0
dinit /dev/fd0
```

To mount a filesystem on the block special device:

```
mount /dev/fd0 /fd0
```

You may now treat the floppy as a QNX filesystem mounted as **/fd0**. To copy files to the floppy, use the **cp** utility.

Tape

To back up/restore from tape, you must make sure the correct driver has been started. This will typically be one of the SCSI drivers described in the *Utilities Reference*.

For example, if you have an Adaptec 1540-compatible SCSI controller with an attached tape drive, the following command will start the driver and register a sequential-access tape unit as **/dev/tape1**:

```
Fsys.aha4scsi fsys -n 1=tape
```

The archive utilities (e.g. **tar**, **pax**) will read and write directly to the tape's block special file. You can't mount a filesystem on this type of block special file.

Currently, the QNX 4 filesystem is limited to handling 512-byte records. To ensure proper streaming performance with tape devices, you may need to increase the blocking factor by using the **-b** *blocking* option to **tar** or **pax**.

For example, the following command:

```
pax -w -b63b -t /dev/td0 .
```

performs a backup of the current directory to tape using the maximum blocking factor with **pax**.

When the AHA 4 driver receives a request to read or write, it does so by starting at the current tape position. If you're starting a new backup, you'll need to erase and rewind the tape.

A number of tape position and control functions are provided by the **tape** utility, which is described in the *Utilities Reference*. For example, the following command would rewind, erase, then rewind the tape in preparation for an archiving procedure:

```
tape rewind erase
```

Removable disk

Removable hard disks are popular in both magnetic and optical formats. Many units use a SCSI interface to the computer, so you may want to consider making your internal fixed disk a SCSI drive (then you won't need a second controller and driver).

Unlike floppies and tapes, a removable hard disk lets you avoid using the archive utilities like **tar**. Instead, you'd likely use **cp** or **pax -rw**to copy your data to a real filesystem on the cartridge. This allows you to recover single files very easily and quickly.

Fixed disk

You can place a second hard disk in your machine for online backups. As an alternative, you could consider making backups to a hard disk on another machine in the network. However, backups across the network are slower than backups to a local hard disk. In this case, you might want to schedule all backups to be done as a **cron** job over night.

Compression

You can use a compression utility to reduce the amount of space required to store data. The amount of compression will depend on the nature of the data you're saving. Some databases containing large amounts of repetitive data may compress up to 90%. Other data might compress less than 10%.

CAUTION: Although compression saves media space, it has two undesirable side effects:

- Compression requires a fair amount of computation and may slow down the process of saving the data.
- You might not be able to recover compressed data if a defect develops during the process. Potentially, all data following a bad block in one part of the compressed data could affect the rest of the data.

You may use the **gzip** or **freeze** utilities to compress your data and the **gunzip** or **melt** utilities to restore it. Both utilities will act on a stream of data as well as on files. This ability to act as a filter lets you connect them to one of the standard archivers via a pipe. For example, we distribute the QNX operating system in compressed form on floppies using **pax**, **freeze**, and **vol**.

Archive examples

Compressed floppy archive

Collect files under **/home** into a **tar** format archive, compress the archive, and then write it out to as many floppies as are needed, adding sequence numbers to the diskettes:

```
pax -w -x ustar /home | freeze | vol -w /dev/fd0
```

Read data off floppies, decompress it, and restore the files:

```
vol -r /dev/fd0 | melt | pax -r
```

UNIX-compatible floppy archive

Save files under **/home/dino** in a **tar** format archive for restoration on a UNIX system:

```
pax -w -t/dev/fd0 -xustar /home/dino
```

Save files under **/home/dino** in a **cpio** format archive for restoration on a UNIX system:

```
pax -w -xcpio -t/dev/fd0 /home/dino
```

Restore data in a **tar** or **cpio** format archive from another UNIX system and place all files under **/usr/unix**:

```
pax -r -s -t/dev/fd0  ",/,/usr/unix/,"
```

Tape archive

Start a new tape archive and save all files to tape:

```
tape rewind erase
pax -w -t/dev/tp0 /
```

Append files that have changed since the date of the file **lastsave** to the end of an existing archive tape. After the save, update the time of **lastsave**:

```
tape forward
touch newsave
find / -newer lastsave | pax -w -t/dev/tp0
mv newsave lastsave
```

Restore all the files on a tape under the directory **/home/dino**:

```
tape rewind
pax -r "/home/dino/*" -t/dev/tp0
```

Cartridge/optical

Copy all files from the filesystem on node 1 to the filesystem on node 2:

```
cp -Rp //1/ //2/
```

In the following example, the disk on node 2 is a very large optical. A full backup is done each Friday and a partial backup of modified files could be done each day of the week:

```
cp -Rp //1/ //2/fri
cp -Rp -a date //1/ //2/mon
cp -Rp -a date //1/ //2/tue
.
.
.
```

You can also do this with **cpio -p**.

Chapter 11

Disk & File Recovery

In this chapter. . .

Introduction

The QNX filesystem achieves high throughput without sacrificing reliability. Although the filesystem is designed to be as robust as possible, 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 QNX filesystem has been designed to tolerate such catastrophes. It is based on the principal that the integrity of the filesystem as a whole should be consistent at all times. While most data is held in the buffer cache and written after only a short delay, critical filesystem data is written immediately. Updates to directories, inodes, extent blocks, and the bitmap are forced to disk to ensure that the filesystem structure on disk is never corrupt (i.e. the data on disk should never be internally inconsistent).

If a crash occurs, you can use the following file maintenance and recovery utilities:

- **fdisk**
- **dinit**
- **chkfsys**
- **dcheck**
- **zap**
- **spatch**

These utilities will let you determine whether any damage was done to files that were open for writing at the time of the crash. These same utilities can also fix such damage, and in many cases will completely restore the filesystem.

Sometimes the damage may be more severe. For example, it's possible that a hard disk will develop a bad block in the middle of a file, or worse, in the middle of a directory or some other critical block.

Again, the utilities we've provided can help you determine the extent of such damage. You can often rebuild the filesystem in such a way as to avoid the damaged areas. In this case, some data will be lost, but with some effort, a large portion of the affected data may be recovered.

Making a recovery floppy

You should always have a *recovery floppy* on hand if, for whatever reason, a machine won't boot from hard disk.

This procedure applies only to QNX systems that were shipped on diskette. If your QNX system came on CD-ROM, refer to the technote in **`/etc/readme/technotes/qnx_install`**, which documents a script for creating a boot floppy.

Before you begin, make sure that you're logged in as **`root`** and that **`Fsys.floppy`** is running.

Now follow these steps:

1. Insert a QNX boot disk in your floppy drive.
2. Copy the image to a temporary file on your hard disk:
   ```
   dd if=/dev/fd0 of=/tmp/floppy_image
   ```
3. Insert a blank floppy in the drive. Format the floppy:
   ```
   fdformat -k0 -z2 /dev/fd0
   ```
4. Copy the image (from your temp file) to the floppy:
   ```
   dd if=/tmp/floppy_image of=/dev/fd0
   ```
5. Run **`dcheck`** to check the new floppy:
   ```
   dcheck /dev/fd0
   ```
   If this fails, retry steps 3 and 4 (**`fdformat`** and **`dd`**); if it fails twice, try a new floppy.

6 Mount the floppy drive filesystem:
```
mount /dev/fd0 /fd
```

7 To make some room on the floppy, remove the following:

- **disktrap** utility (14K):
```
rm /fd/bin/disktrap
```

- All the disk drivers from **/fd/bin** that aren't needed for this machine.

8 Now copy these useful utilities to **/fd/bin**:
```
cp /bin/sin /fd/bin/sin
cp /bin/zap /fd/bin/zap
cp /bin/rm /fd/bin/rm
cp /bin/ls /fd/bin/ls
cp /bin/spatch /fd/bin/spatch
cp /bin/chkfsys /fd/bin/chkfsys
cp /usr/bin/elvis /fd/bin/elvis
```

9 Create a **/etc** directory:
```
mkdir /fd/etc
```

10 Copy a **termcap** file:
```
cp /etc/termcap /fd/etc/termcap
```
You should also edit the **termcap** file and remove the entries you won't need. The only entry you'll need is the one for QNX.

11 Now we need to create two links:
```
cd /fd/bin
ln -s elvis vi
ln -s fcat melt
```

12 Finally, you'll need to modify the system initialization file (**/fd/etc/config/sysinit**) so that it now contains these lines:

```
Dev -n 10 &
Dev.con -n 4 -O 256 &
reopen /dev/con1
export PATH=/ram:.:/bin:/usr/bin
export HOME=/
dinit /dev/ram
mount /dev/ram /ram
prefix -A /pipe=/ram
prefix -A /tmp=/ram
fcat /util.tar.F | pax -vr
cp /bin/esh /ram/sh
melt -z </etc/logo.F
rtc hw
echo Welcome to QNX 4.25
ontty /dev/con1 /bin/sh
ontty /dev/con2 /bin/sh
```

That's it! Keep your recovery floppy in a safe place. If and when you ever need to use it, simply insert the floppy in a dead machine and power on — the machine will boot QNX from the floppy.

Overview of QNX disk structure

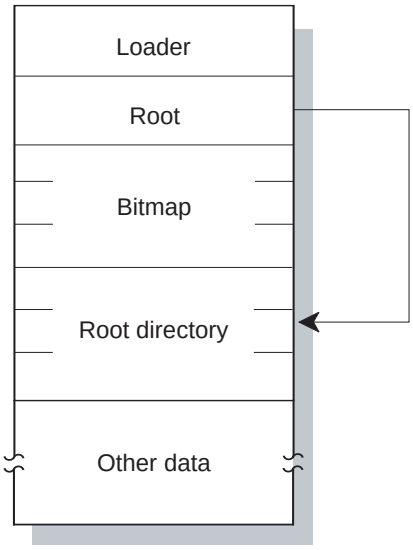
In this section, we describe how the QNX filesystem stores data on a disk. Reading this section should help you recognize and possibly correct filesystem damage if you ever have to rebuild a filesystem.

If you have a C development package, the header file **<sys/fsys.h>** contains the definitions for all terms used in this section.

For an overall description of the QNX filesystem, see the Filesystem Manager chapter in *System Architecture*.

Partition components

A QNX filesystem may be an entire disk (in the case of floppies) or it may be one of many partitions on a hard disk. Within a disk partition, a QNX filesystem contains the following components:



The following blocks are always found, in this order, on a QNX disk partition:

- loader
- root
- bitmap
- root directory
- other data

Loader block

The loader block is the first block of a QNX partition. It contains the bootstrap loader that loads the QNX OS into memory.

Root block

The root block is the second block of a QNX partition. It contains the directory entry for the root (/), the inode entries for the inode file, and a label field.

Bitmap blocks

Several consecutive blocks follow the root block. The bitmap blocks form the bitmap for the QNX partition. One bit exists for each block on the partition, thus one bitmap block will be used for every 4096 disk blocks (corresponding to 2M of disk space).

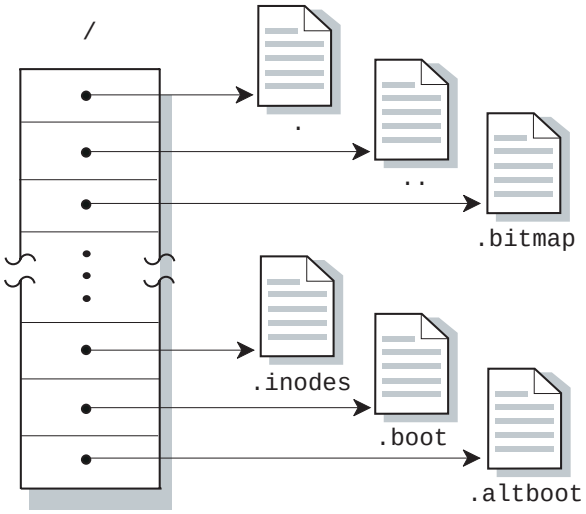
If the value of a bit is zero, its corresponding block is unused. Unused bits at the end of the last bitmap block (for which there are no corresponding disk blocks) are turned on.

Bit assignments start with the least-significant bit of byte 0 of the first bitmap block — which corresponds to QNX block #1.

Root directory

The root directory follows the bitmap blocks. The root directory is a "normal" directory (see the "Directories" section). It is initially created by the **`dinit`** utility with enough room for 32 directory entries (4 blocks).

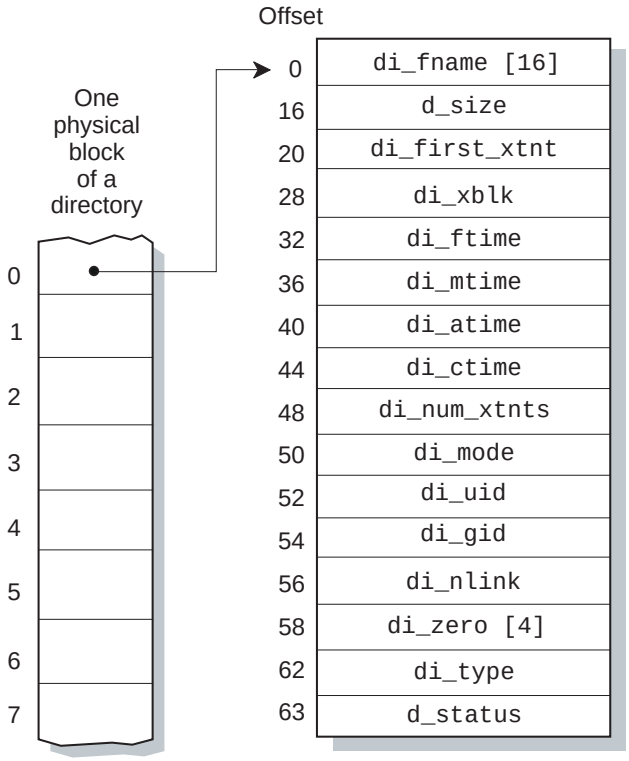
As the following illustration shows, the root directory (`/`) contains directory entries for several special files that always exist in a QNX filesystem. The **`dinit`** utility creates these files when the filesystem is first initialized.



File:	Description:
/.	A link to the **/** directory
/..	Also a link to the **/** directory
/.bitmap	Represents a read-only file consisting of the bitmap blocks.
/.inodes	A normal file of at least one block on a floppy/RAM disk and 16 blocks on other disks, **/.inodes** is a collection of **inode** entries. The first entry is reserved and used as a signature/info area. The first bytes of the **.inode** file are “IamTHE.inodeFILE”.
/.boot	Represents an OS image file that will be loaded into memory during the *standard* boot process. This file will be of zero length if no boot file exists.
/.altboot	Represents an OS image file that will be loaded into memory during the *alternate* boot process. This file will be of zero length if no alternate boot file exists.

Directories

A directory is simply a file that has special meaning to the filesystem. A directory file contains a collection of directory entries as shown in the following illustration:



The type of directory entry is determined by the bits in the **`d_status`** field, as follows:

Bit 3 (`_FILE_LINK`)	Bit 0 (`_FILE_USED`)	Comment:
0	0	unused directory entry
0	1	normal, used directory entry
1	0	link to an entry in **`/.inodes`** (which should be used)

continued...

Bit 3 (_FILE_LINK)	Bit 0 (_FILE_USED)	Comment:
1	1	invalid

The first directory entry is always for the file "**.**" and includes a directory signature ("I♥QNX"). The hexadecimal equivalent of the ♥ character is 0x03. This entry refers to the directory itself by pointing to the entry within the parent directory that describes this directory.

The second entry is always for the "**..**" file. This entry refers to the parent directory by pointing to the first block of the parent directory.

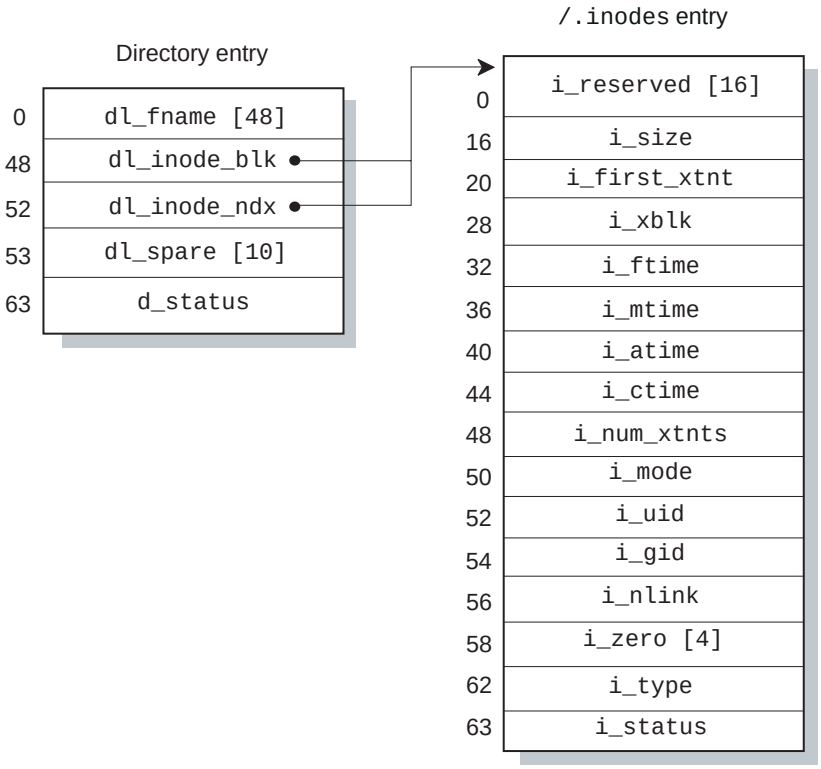
Every directory entry either defines a file or points to an entry within the **/.inodes** file. Inode entries are used when the filename exceeds 16 characters or when two or more names are linked to a single file.

The first extent (if any) of a file is described in the directory/inode entry. Additional file extents require a linked list of extent blocks whose header is also in the directory/inode entry. Each extent block in the chain points to between 1 and 60 extents.

Links

Files with names greater than 16 characters and *links* to other files are implemented with a special form of directory entry. These entries are identified with the **_FILE_LINK** bit (0x08) of the **d_status** field being set.

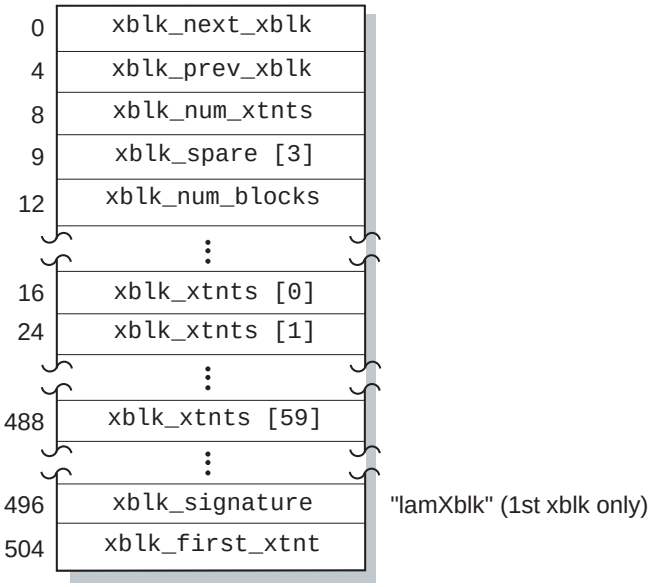
For these files, a portion of the directory entry is moved into the **/.inodes** file.



Extent blocks

Extent blocks are used for any file that has more than a single extent. The directory entry **`di_xblk`** *points* to one of these extent blocks, which in turn defines where the second and subsequent extents are to be found.

An extent block is exactly one 512-byte disk block with the following form:



Each extent block contains:

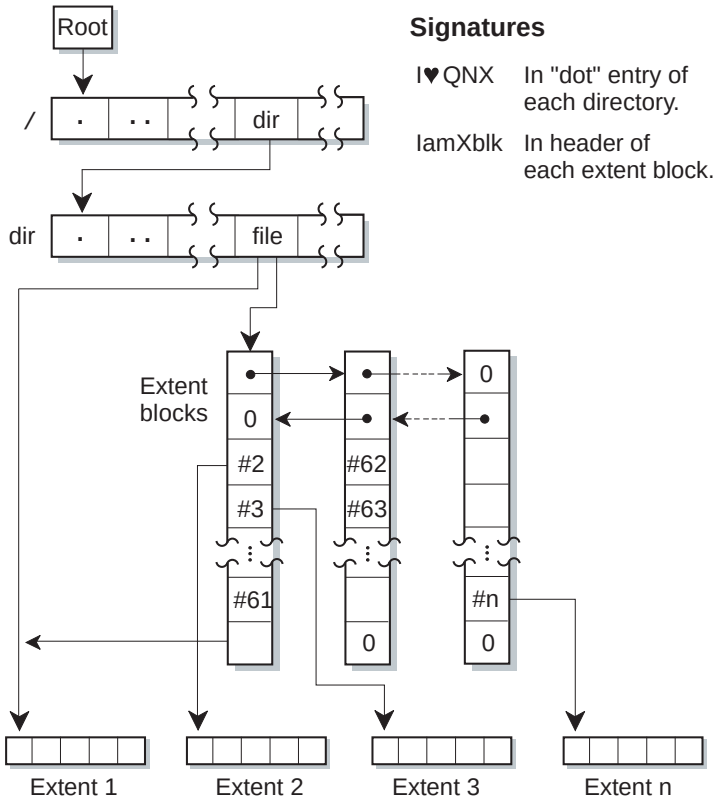
- forward/backward pointers
- a count of extents
- a count of all the blocks in all the extents defined by this extent block
- pointers and block counts for each extent
- a signature (“IamXblk”)

The first extent block also contains a redundant pointer to the first file extent (also described within the directory/inode entry). This lets you recover all data in the file by locating this block alone.

Files

Files or file extents are groupings of blocks described by directory/inode entries; they have no structure imposed on them by the QNX filesystem.

Most files in QNX have the following overall structure:



File maintenance utilities

fdisk

The **fdisk** utility creates and maintains the *partition block* on a hard disk. This block is compatible with other operating systems and may be maintained by other OS versions of **fdisk** (although ours has the advantage of recognizing QNX-specific information). If the partition loader is missing or damaged, **fdisk** can create it.

☞ We recommend you keep a hard copy of the partition table information for every disk in your network.

dinit

The **dinit** utility creates (but **Fsys** maintains) the following:

- boot block
- root block
- bitmap blocks
- root directory
- **/.inodes** file

chkfsys

The **chkfsys** utility is your principal filesystem maintenance tool. This utility:

- checks the directory structure of an entire disk partition, reports any inconsistencies, and fixes them, if possible
- verifies overall disk block allocation
- writes a new **/.bitmap**, upon your approval

The **chkfsys** utility assumes that the root block is valid. If the root block isn't valid, **chkfsys** will complain and give up — you'll need to try restoring the root block with the **dinit** utility.

dcheck

The **dcheck** utility verifies that a disk has been correctly formatted by attempting to read every block on the drive. When the **-m** option is specified, **dcheck** removes any bad blocks from the disk allocation bitmap (**/.bitmap**).

If the file **/.bad_blks** is found, **dcheck** will update the bitmap and recreate the **/.bad_blks** file. You can run **dcheck** a few times to increase your chances of bad blocks being recognized and added to the **/.bad_blks** file.

zap

The **zap** utility lets **root** remove files or directories from the filesystem without returning the used blocks to the free list. You might do this for several reasons, including the following:

- the directory entry is damaged
- two files occupy the same space on the disk (an error)

Recovering a zapped file

If you zapped a file in error, it's sometimes possible to recover the zapped file using the **zap** utility with the **-u** option *immediately* after the deletion. You can recover a zapped file using **zap** under these conditions:

- the directory entry for that (now deleted) file must *not* be reused
- the disk blocks previously used by the file must *not* be reassigned to another file

spatch

The **spatch** utility lets you browse the raw disk and patch minor problems. You can sometimes cure transient disk problems by reading and writing the failing block with **spatch**.

Disk recovery procedures

Using chkfsys

The **chkfsys** utility is your principal tool for checking and restoring a potentially damaged filesystem. It can identify and correct a host of minor problems as well as verify the integrity of the entire disk system as a whole.

Normally, **chkfsys** requires that the filesystem be idle and that no files be currently open on that device. You'll have to shut down any processes that have opened files or that may need to open files while **chkfsys** is running.

To run **chkfsys** on a mount point, you'd simply type:

```
chkfsys /
```

The utility scans the entire disk partition from the root down, building an internal copy of the bitmap and verifying the consistency of all files and directories it finds in the process.

When it has finished processing all files, **chkfsys** compares the internal bitmap to the bitmap on the disk. If they match, **chkfsys** is finished. If any discrepancies are found, **chkfsys** will — upon your approval — rewrite the bitmap with data consistent with the files it was able to find and verify.

In addition to verifying block allocation (bitmap), **chkfsys** attempts to fix any problems it finds during the scan. For example, **chkfsys** can:

- "unbusy" files that were written during a crash
- fix the file size in a directory entry to match the real data

When to run chkfsys

It's a good idea to run **chkfsys** as part of your regularly scheduled maintenance procedures — this lets you verify that the data on your disk is intact. For example, you might consider running **chkfsys** on your network servers every time they boot. An automated check on the filesystem at boot time guarantees that **chkfsys** will attempt to fix any problems it finds during the scan. To automate this process, add **chkfsys** to the server's **sysinit.***node* file.

It's especially important to run **chkfsys** after a system crash, power outage, or unexpected system reboot so that you can identify whether any files have been damaged. The **chkfsys** utility checks the "clean" flag on the disk to determine whether the system was in a consistent state at the time.

The clean flag is stored on disk and is maintained by the system. The flag is turned off whenever a file is opened for update and is turned on after all open files have been closed and the associated data has been

flushed from cache to disk. When the clean flag is set, **chkfsys** assumes that the filesystem is intact. If **chkfsys** finds the clean flag off, it tries to fix the problem.

The **chkfsys** utility supports a **-u** option, which overrides a set clean flag and tells **chkfsys** to run unconditionally. You might want to override the clean flag when:

- **dcheck** discovers bad blocks
- files have been deleted or zapped intentionally
- you want to force a general sanity check

Using chkfsys on a live system

The **chkfsys** utility normally requires exclusive use of the filesystem to provide a *comprehensive* verification of the disk.

CAUTION: There is some risk to running **chkfsys** on a live system — both **chkfsys** and the filesystem are reading and possibly writing the same blocks on the disk. Also, the filesystem has internal cached data about files and directories that can't be updated when **chkfsys** makes a change. But static changes, in place, on files or directories that **Fsys** doesn't currently have opened will *probably* not cause problems.

If you're running an application that can't afford downtime or you couldn't run **chkfsys** because files were open for updating, try to run **chkfsys** with the **-f** option:

```
chkfsys -f /dev/hd0t77
```

This invokes a special read-only mode of **chkfsys**. It will give you a feeling for the overall sanity of your filesystem.

Recovering from a bad block in the middle of a file

Hard disks occasionally develop bad blocks as they age. In some cases, you might be able to recover most or even all the data in a file containing a bad block.

Some bad blocks are the result of power failures or of weak media on the hard disk. In these cases, sometimes simply reading then rewriting a block will "restore" the block for a short period of time. This may allow you to copy the entire file somewhere else before the block goes bad again. This procedure certainly can't hurt, and is often worth a try.

To examine the blocks within a file, you use the **spatch** utility. When you get to a bad block, **spatch** should report an error, but it may have actually read a portion of "good" bytes from that block. Writing that same block back will often succeed.

At the same time, **spatch** will rewrite a correct CRC (Cyclic Redundancy Check) that will make the block good again (but with possibly incorrect data).

You can then copy the entire file somewhere else, and then **zap** the previously damaged file. To complete the procedure, you mark the marginal block as bad (by adding it to the **/.bad_blks** file), then run **chkfsys** to recover the remaining good blocks.

If this procedure fails, you can use the **spatch** utility to copy as much of the file as possible to another file, and then **zap** the bad file and run **chkfsys**.

What to do if your system will no longer boot

If a previously working QNX system suddenly stops working and will no longer boot, then one of the following may have occurred:

- the hardware has failed or the data on the hard disk has been damaged

- someone has either changed/overwritten the boot file or changed the system initialization file — these are the two most common scenarios

The following steps can help you identify the problem. Where possible, corrective actions are suggested.

Step 1 — Try booting from floppy or across the network

If you have a network to boot over, try booting your machine over the network. Once the machine is booted, you'll need to log in as **`root`** and then start up a local filesystem:

```
Fsys &
```

If you don't have a network, you'll need to boot from your recovery floppy (described earlier in this section) or from the QNX boot floppy that was used to install your system onto the hard disk. The filesystem will already be running in this case, and you'll be logged in as **`root`**.

Step 2 — Start the hard disk driver

You now have to start the appropriate hard disk driver. For example, to start a driver for an Adaptec series 4 SCSI adapter, you would type:

```
Fsys.aha4scsi &
```

If you're using another type of driver, enter its name instead.

This should create a block special file called **`/dev/hd0`** that represents the entire hard disk.

Step 3 — Run `fdisk`

Running the **`fdisk`** utility will immediately give you useful information about the state of your hard disk.

The **`fdisk`** utility might report one of several types of problems:

Problem:	Probable cause:	Remedy:
Error reading block 1	Either the disk controller or the hard disk itself has failed.	If the disk is good, replacing the controller card *might* let you continue using the disk. Otherwise, you'll have to replace the hard drive, reinstall QNX, and restore your files from backup.
Wrong disk parameters	Your hardware has probably "lost" its information about this hard drive — likely because the battery for the CMOS memory is running low.	Rerunning the hardware *setup* procedure (or the programmable option select procedure on a PS/2) will normally clear this up. Of course, replacing the battery will make this a more permanent fix.
Bad partition information	If the disk size is reported correctly by **`fdisk`**, but the partition information is wrong, then the data in block 1 of the physical disk has somehow been damaged.	Use **`fdisk`** to recreate the correct partition information. It's a good idea to write down or print out a hard copy of the correct partition information in case you ever have to do this step.

Step 4 — Mount the partition and the filesystem

At this point, you have verified that the hardware is working (at least for block 1) and that a valid partition is defined for QNX. You now need to create a block special file for the QNX partition itself and to mount the block special file as a QNX filesystem:

```
mount -p /dev/hd0 /dev/hd0t77 /hd
```

This should create a volume called **/dev/hd0t77**. Depending on the state of the QNX partition, the **mount** may or may not fail. If the partition information is correct, there shouldn't be any problem. Since the root (**/**) already exists (on a floppy or on a remote disk on the network), we've mounted the local hard disk partition as a filesystem with the name **/hd**.

Your goal now would be to run the **chkfsys** utility on the disk to examine — and possibly fix — the filesystem.

If you booted from floppy and you don't suspect there's any damage to the filesystem on your hard disk (e.g. the system was unable to boot because of a simple error introduced in the boot file or system initialization file), you can change the root prefix to your hard disk partition at this point with the following command, which will resume normal operation of the system:

```
/hd/bin/prefix -R /=/hd/
```

If you run this command, you can skip the rest of this section.

If the mount fails...

If the **mount** fails, the first portion of the QNX partition is probably damaged (since **Fsys** will refuse to mount what it considers to be a corrupted filesystem).

In this case, you can use the **dinit** utility to overlay enough good information onto the disk to satisfy **Fsys**:

```
dinit -hr /dev/hd0t77
```

The **-r** option tells **dinit** to rewrite:

- the root block
- the bitmap (with *all* blocks allocated)
- the constant portions of the root directory

You should now be able to reissue the **mount** command and once again try to create a mount point for a QNX filesystem called **/hd**.

After doing this, you'll need to rebuild the bitmap with **chkfsys**, even on a good partition.

Step 5 — Run chkfsys

At least a portion of your QNX filesystem should now be accessible. You can use **chkfsys** to examine the filesystem and recover as much data as possible.

If the hard disk is mounted as **/hd** (e.g. the machine boots from floppy), enter:

```
/hd/bin/chkfsys /hd
```

If the hard disk is mounted as **/** (e.g. a network boot), enter:

```
chkfsys /
```

In either case, you should make note of any problems reported and allow **chkfsys** to fix as much as it can. What you do next depends on the result of running **chkfsys**.

If the disk is unrecoverable

If, for any reason, your disk is completely unrecoverable, read the next section, "Recovering lost files and directories." In some cases, you may need to reinstall QNX from floppy and restore your disk from your backup files.

If significant portions of the filesystem are irreparably damaged, or important files are lost, then restoring from backup might be your best alternative.

If the filesystem is intact

If your filesystem is intact, yet the machine still refuses to boot from hard disk, then either of the following is probably damaged:

- the partition loader program in physical block 1

- the QNX loader in the first block of the QNX partition

To rewrite a partition loader, use **fdisk**:

```
fdisk /dev/hd0 loader
```

To rewrite the QNX loader, use **dinit**:

```
dinit -b /dev/hd0t77
```

You should now be able to boot your system.

Recovering lost files and directories

You may sometimes find that files or directories have been completely lost due to disk corruption. If after running **chkfsys** you know that certain key files or directories were *not* recovered, then you might be able to use **spatch** to recover some or all of this data.

Before attempting this, you should first familiarize yourself with the details of a QNX filesystem (see "Overview of QNX disk structure" in this chapter). You should also study the documentation for the **spatch** utility in the *Utilities Reference*.

Appendix A

QNX Console & Keyboard Conventions

In this appendix. . .

This appendix describes QNX's standard console and keyboard conventions *in text mode*. For information on the Photon graphical environment, see the Photon *User's Guide*.

Note that some keys may behave differently from how they're described here, depending on how your system has been configured.

For details on the utilities that control QNX consoles, see the following in your *Utilities Reference*:

- **`Dev`**
- **`Dev.ansi`**
- **`Dev.con`**
- **`sh`**
- **`stty`**

Entering line-oriented input

Line-editing keys

Many applications run in *edited* mode. If an application runs in this mode, you can use the following keys for entering line-oriented input:

If you want to:	Press this key:
Move the cursor left one position	← (left arrow key)
Move the cursor right one position	→ (right arrow key)
Move the cursor to the beginning of the line	Home
Move the cursor to the end of the line	End

continued...

If you want to:	Press this key:
Delete the character to the left of the current cursor position	Backspace or Rubout or ← (keypad arrow). Note that pressing this key generates a **7F** hex (ASCII Rubout), not a **08** hex.
Delete the character at the current cursor position	Del
Delete all characters on the current line	Ctrl – U
Toggle between insert mode and typeover mode (default is insert)	Ins Note that if you're in typeover mode and you submit a line, you'll be returned to insert mode.
Submit a line of input	Enter

The QNX shell has additional input editing commands. For more information, see **sh** in the QNX *Utilities Reference*.

Note also that your keyboard may not behave as indicated if:

- you're working with an application that has complex requirements for user interaction — the application may take control over how the keyboard works
- you're working at an attached terminal — the terminal may have keyboard limitations.

Max length of an input line

The maximum length of an input line is 256 characters. Application programs can impose lower limits.

Entering long input lines

If you enter a single line of input that's too long for your screen to display as a single line, the line may be displayed as several screen lines. But the line still behaves as a single input line when you use the standard line-editing keys.

For more information on edited mode, see *System Architecture*, Chapter 6, The Device Manager.

Recalling commands

The shell lets you recall commands that you've previously entered, then re-execute them. These commands are maintained by the shell in a buffer.

If you want to move:	Press this key:
Backward through the buffer	↑ (up arrow key)
Forward through the buffer	↓ (down arrow key)

Switching virtual consoles

The display adapter, the screen, and the system keyboard are collectively referred to as the *console*. To let you interact with several applications all at once, QNX permits multiple sessions to be run concurrently on consoles by means of *virtual consoles*. These virtual consoles are usually named **`/dev/con1`**, **`/dev/con2`**, etc.

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 the following keychords.

If you want to see the:	Press:
Next active console	Ctrl – Alt – Enter *or* Ctrl – Alt – + (keypad plus key)
Previous active console	Ctrl – Alt – - (keypad minus key)

You can also "jump" to a specific console using the Ctrl – Alt – *n* keychord, where *n* is the numeric digit that represents the console number of a virtual console.

If you want to see:	Press:
`/dev/con1`	Ctrl – Alt – 1
`/dev/con2` (if available)	Ctrl – Alt – 2
⋮	⋮
`/dev/con10` (if available)	Ctrl – Alt – 0

You can disable keyboard console switching with the **`stty +noswitch`** command.

For more information on the QNX console, see *System Architecture*, Chapter 6, The Device Manager.

Using multiple consoles

The system administrator can specify how many virtual consoles are supported on your machine by specifying an argument to the console driver process (**`Dev.con`**) when it's started.

The administrator can also specify the program — if any — that's initially launched on each console. By default, the terminal initialization utility (**`tinit`**) launches a **`login`** on the first console only, but will be "armed" to launch a **`login`** on any other console on which you press a key. This means that while console 1 is always

available, any other given console won't be used unless you specifically switch to that console and press a key.

To start a **login** on an unused console:

1. Switch to the unused console via Ctrl – Alt – *n*
2. Press any key. A **login** will be launched.

You can now access the console via any of the cyclical console-switching keychords (e.g. Ctrl – Alt – +) described in the previous section.

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

Changing the console fonts

Depending on your video hardware, the console driver (**Dev.con**) may support a variety of screen fonts. The available fonts are numbered from 0 to *n*. When QNX boots, it defines font 0 as a 25×80 text font. If you're using an EGA or VGA video adapter, QNX also defines font 1 as a 43×80 text font (EGA) or a 50×80 text font (VGA).

To inform **Dev.con** about new fonts or to redefine existing fonts, you use the **cfont** utility. You can use this utility to provide fonts of different sizes or fonts that contain alternate character sets.

To change the font associated with the current console, use the following keychords:

If you want to choose the:	Press:
Next font (up to *n*)	Ctrl – Alt – **>**
Previous font (down to 0)	Ctrl – Alt – **<**

QNX keeps track of the font being used by each console. All consoles initially display font 0. You can disable keyboard font changing with the **stty +noresize** command.

Suspending and resuming output

If you want to:	Press:
Suspend the display of output	Ctrl – S
Resume the display of output	Ctrl – Q

Killing a process

If you need to kill the process currently running on the console, press Ctrl – C or Ctrl – Break. The system will attempt to kill the process.

Invoking the system debugger

QNX comes with a low-level system debugger that lets you set breakpoints in programs, display and edit memory, disassemble code, and examine I/O ports. If it has been built into your OS image, you can invoke this debugger with the following keychord:

Ctrl – Alt – Esc

You can disable this keychord with the **stty +nodebug** command.

CAUTION: Don't use this debugger in a multiuser environment, because it disables interrupts and freezes the entire system — it's intended only for low-level debugging. For more information on this debugger, see **Debugger** in the QNX *Utilities Reference*. For information on general debugging, see the Watcom Debugger *User's Guide*.

Rebooting

To reboot your computer, use this keychord:

Ctrl – Alt – Shift – Del

You can disable this keychord with the **`stty +noboot`** command.

CAUTION: Before entering this command, make sure that no applications or utilities are running on your computer. If there are, files may be left open. Moreover, if you reboot when a critical update is in progress, it's possible that the filesystem would require maintenance (see the section on "Filesystem robustness" in *System Architecture*; see also the chapter on Disk & File Recovery in this guide).

International keyboards

Some keyboard layouts — the French and German layouts, for example — use accent keys which, by themselves, don't generate a character. QNX treats these keys as "dead" keys. Pressing a dead key, followed by a second key, modifies the second key, creating an accented character. For example, to create the **Ü** character, you press ¨ followed by Shift – U.

This dead key processing provides typists with a familiar method of composing characters.

Note that you can also generate composed characters by:

1. pressing and releasing Alt
2. typing two characters.

The keyboard at a glance

If you want to:	Press:
Move the cursor to the left	←
Move the cursor to the right	→
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 or ← (keypad arrow key)
Delete the character at the cursor	Del
Delete all characters on a line	Ctrl – U
Toggle between insert/typeover modes	Ins
Submit a line	Enter
Recall a command	↑ or ↓
Switch to the next virtual console	Ctrl – Alt – Enter *or* Ctrl – Alt – +
Switch to the previous virtual console	Ctrl – Alt – -
Switch to a specific console	Ctrl – Alt – *n*
Choose the next font	Ctrl – Alt – **>**
Choose the previous font	Ctrl – Alt – **<**
Suspend display of output	Ctrl – S
Resume display of output	Ctrl – Q
Attempt to kill a process	Ctrl – C or Ctrl – Break

continued...

If you want to:	Press:
Indicate end of input (EOF)	Ctrl – D
Invoke the system debugger	Ctrl – Alt – Esc
Reboot your computer	Ctrl – Alt – Shift – Del

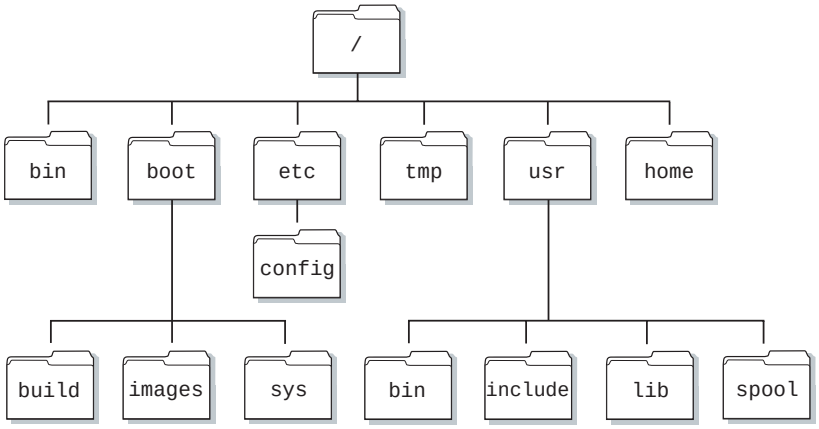
Appendix B

Where QNX Files Are Found

In this appendix. . .

A typical directory structure

QNX modules are stored according to the kind of services they provide. The following diagram gives an overview of the general directory structure in a typical QNX installation:



Summary of file locations

To find:	Look in:
system executables	**/bin**
OS image **Makefile**	**/boot**
build files to make images (the **make** utility reads these)	**/boot/build**
OS image files	**/boot/images**
system processes needed at boot time	**/boot/sys**
initialization and other files	**/etc**
sysinit and configuration files	**/etc/config**

continued...

To find:	Look in:
software release information files	**/etc/readme**
technical notes	**/etc/readme/technotes**
default location for temporary files	**/tmp**
more executables	**/usr/bin**
header (***.h**) files for the C compiler	**/usr/include**
libraries for the C compiler	**/usr/lib**
terminal capabilities files	**/usr/lib/terminfo**
work files for system spoolers and queues	**/usr/spool**
a user's home directory	**/home/***username*

Index

!

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z