QNX® Neutrino® Realtime Operating System

Building Embedded Systems

For targets running QNX® Neutrino® 6.4

Published under license by:

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

Electronic edition published 2009.

Contents

List of Figures

About This Book

What you'll find in this guide

The *Building Embedded Systems* guide is intended for developers who are building embedded systems that will run under the QNX Neutrino RTOS.

QNX Neutrino runs on several processor families (e.g. PowerPC, MIPS, ARM, SH-4, x86). For information on getting started with Neutrino on a particular board, refer to the appropriate BSP (Board Support Package) documentation for your board.

This guide is organized around these main topics:

Topic	Chapter(s)
Getting the big picture	Overview of Building Embedded Systems
Getting started with your board support package	Working with a BSP
Making an image	Making an OS Image
Preparing your target	Writing an IPL Program; Customizing Image Startup Programs; Customizing the Flash Filesystem; Sample Buildfiles
Dealing with hardware issues	System Design Considerations
Terms used in QNX docs	Glossary

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

For information about programming in Neutrino, see *Getting Started with QNX Neutrino: A Guide for Realtime Programmers* and the Neutrino *Programmer's Guide*. If you plan to use the Photon microGUI in your embedded system, refer to the "Photon in Embedded Systems" appendix in the Photon *Programmer's Guide*.

Typographical conventions

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

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	**`something you type`**
Keyboard keys	Enter
Program output	**`login:`**
Programming constants	NULL
Programming data types	**`unsigned short`**
Programming literals	`0xFF`, **`"message string"`**
Variable names	*stdin*
User-interface components	**Cancel**

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

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

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

Notes point out something important or useful.

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

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

Note to Windows users

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

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

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

Chapter 1

Overview of Building Embedded Systems

In this chapter. . .

Introduction

In this chapter, we'll take a "high-level" look at the steps necessary to build a complete Neutrino-based embedded system, with pointers to the appropriate chapters for the lower-level details.

First we'll see what a Neutrino system needs to do in order to run. Then we'll look at the components and how they operate. Finally, we'll do an overview of the steps you may need to follow when customizing certain portions.

From the software perspective, the following steps occur when the system starts up:

1. Processor begins executing at the reset vector. The Initial Program Loader (IPL) locates the OS image and transfers control to the startup program in the image.
2. Startup program configures the system and transfers control to the Neutrino microkernel and process manager (**`procnto`**).
3. The **`procnto`** module loads additional drivers and application programs.

After we look at the software aspects in some more detail, we'll consider the impact that the hardware has on this startup process.

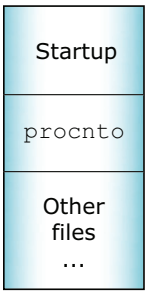
The role of the IPL

The first step performed by the software is to load the OS image. This is done by a program called the *Initial Program Loader* (IPL).

The IPL's initial task is to minimally configure the hardware to create an environment that will allow the startup program, and consequently the Neutrino microkernel, to run. Specifically, this task includes at least the following steps:

1. Start execution from the reset vector.
2. Configure the memory controller, which may include configuring chip selects and/or PCI controller.
3. Configure clocks.
4. Set up a stack to allow the IPL lib to perform OS verification and setup (image download, scan, setup, and jump).

The IPL is described in detail in the chapter on Writing an IPL Program.



An OS image loaded by the IPL.

Warm-start and cold-start IPL

There are two general types of IPL: warm-start and cold-start. Warm-start IPL is typically invoked by a ROM-monitor or BIOS; some aspects of the hardware and processor configuration will have already been set up.

With cold-start IPL, on the other hand, nothing has been configured or initialized — the CPU and hardware have just been reset. Naturally, the work that needs to be done within a warm-start IPL will be a subset of the work required in a cold-start IPL.

We'll approach the discussion of the IPL's responsibilities starting at the end, describing the goal or final state that everything should be in just before the first component of the image is started. Then we'll take a look at the steps necessary to get us to that final state.

Depending on the design of your target, you may have to take a number of steps, ranging from none (e.g. you're running on a standard platform with a ROM monitor or BIOS, and have performed a warm-start IPL via disk or network boot; the boot ROM has done all the work described below for you) to many (e.g. you have a custom embedded system without firmware and the image is stored on a specialized piece of hardware).

The final state (just before the first component of the image is started) is characterized by the following:

- The memory controller has been configured to give access to the memory present on the system.
- Minimal hardware configuration has been performed (e.g. chip selects to map EPROMs have been programmed).
- The entire image is now located in linearly addressable memory.
- The first part of the image, the startup code, is now in RAM. (Note that the startup code is relatively small and that the RAM area is reclaimed when the startup code is finished.)

Either the IPL or the BIOS/ROM monitor code is responsible for transferring the image to linearly addressable memory. The OS image must have been built in a format

that the IPL or ROM monitor code understands so that it can know where to place the image in memory and to what address to pass control after the image has been loaded.

For example, an IBM PC BIOS system typically loads a raw binary and then jumps to the first address. Other systems may accept an image in ELF format, using the ELF header information to determine the location to place the image as well as the starting address. Refer to the documentation that came with your hardware to find out what image formats the IPL code can accept.

Once the IPL has located the image, and the entire image is now in linearly addressable memory, control is transferred to the startup program. At that point, the IPL is done and is out of the picture.

The role of the startup program

The second step performed by the software is to configure the processor and hardware, detect system resources, and start the OS. This is done by the *startup program*. (For details, see the chapter on Customizing Image Startup Programs.)

While the IPL did the bare minimum configuration necessary to get the system to a state where the startup program can run, the startup program's job is to "finish up" the configuration. If the IPL detected various resources, it would communicate this information to the startup program (so it wouldn't have to redetect the same resources.)

To keep Neutrino as configurable as possible, we've given the startup program the ability to program such things as the base timers, interrupt controllers, cache controllers, and so on. It can also provide *kernel callouts*, which are code fragments that the kernel can call to perform hardware-specific functions. For example, when a hardware interrupt is triggered, some piece of code must determine the source of the interrupt, while another piece of code must be able to clear the source of the interrupt.

Note that the startup program does *not* configure such things as the baud rate of serial ports. Nor does it initialize standard peripheral devices like an Ethernet controller or EIDE hard disk controller — these are left for the drivers to do themselves when they start up later.

Once the startup code has initialized the system and has placed the information about the system in the *system page* area (a dedicated piece of memory that the kernel will look at later), the startup code is responsible for transferring control to the Neutrino kernel and process manager (**`procnto`**), which perform the final loading step.

Startup's responsibilities

Let's take a look at the overall responsibilities and flow of the startup code:

1 Copy and decompress the image, if necessary.

2 Configure hardware.

3 Determine system configuration.

4 Start the kernel.

Copying and decompressing the image

If the image isn't in its final destination in RAM, the startup code copies it there. If the image is compressed, the startup code automatically decompresses the image. Compression is optional; you can create an image file that isn't compressed, in which case the startup code won't bother trying to decompress it.

Configuring the hardware

The main task here is to set up the minimum required to be able to determine the system configuration (and then perform the system configuration).

The details of what needs to be configured during the hardware configuration phase depend on your particular hardware.

Determining system configuration

Depending on the nature of the embedded system, you may wish to dynamically determine the configuration on startup or (in the case of a deeply embedded system) simply "hardcode" the configuration information.

Regardless of the source of the information, the configuration part of the startup code needs to store this information into a set of well-defined data structures that the OS will then look at when it starts. Collectively known as the *system page area*, these data structures contain information about:

- memory configuration
- hardware device configuration
- processor type
- time of day

Establishing callouts

To keep the Neutrino kernel as portable as possible (not only to different processors, but also to different hardware configurations of those processors), a number of callouts must be supplied by the startup code. Not all of the callouts require that *you* write code — we have a library that provides many of these.

The following classes of callout functions can be provided for Neutrino:

- debug interface
- clock/timer interface
- interrupt controller interface
- cache controller interface
- power management
- miscellaneous

The callouts are described in detail in the chapter on Customizing Image Startup Programs.

Starting the OS

The final step that the startup code performs is to start the operating system.

The startup library

If all of the above sounds like a lot of work, well, it is! Note, however, that we've provided source code for some common startup programs and have created a library that performs most of the above functions for you.

If you have one of the many platforms that we support, then you don't have to do any of this work — we've already done it for you.

To find out what processors and boards we currently support, please refer to the following sources:

- the **boards** directory under *bsp_working_dir*/**src/hardware/startup/boards**.
- QNX docs (BSP docs as well as **startup-*** entries in the *Utilities Reference*).

If you have a nonstandard embedded system, you can look at the source for the system that most closely resembles yours and "clone" the appropriate functionality from the examples provided.

This issue is discussed in detail in the chapter on Customizing Image Startup Programs.

The role of Neutrino

The third step performed by the software is to start any executables that you want to be running. The OS does this by reading and processing information stored in the *startup script* — a sequence of commands stored within the image. The format of the startup script, as well as the *buildfile* that it's part of, is documented in detail in a variety of places in this guide:

- Making an OS Image chapter — describes the steps required to build a Neutrino-based system, including discussions of the script file and buildfile.
- Sample Buildfiles appendix in this guide — describes common "tricks" used within the buildfile and also contains complete examples of sample configurations.
- **mkifs** doc — describes the **mkifs** utility, which is used to create the image from the description passed to it in the buildfile. See the *Utilities Reference* for details.
- Building OS and Flash Images chapter in the IDE *User's Guide* — describes the how the OS and flash images are created in the IDE.

Basically, the OS processes the startup script file, which looks like a shell script. In the startup script file, you'd specify which executables should be started up (and their order), the command-line options that they should run with, and so on.

Hardware aspects

From the hardware point of view, the following components form the system:

- processor
- source of initialization and configuration info
- storage media
- I/O devices

Choice of processor

We support the following processor families:

- ARM (including XScale)
- MIPS
- PowerPC
- SH-4
- x86

At the "altitude" of this high-level discussion, the choice of processor is irrelevant — the same basic steps need to be performed regardless of the particular CPU.

Source of initialization and configuration

When the processor (re)starts, it must be able to execute instructions. This is accomplished by having some kind of nonvolatile storage media placed at the processor's reset vector. There is, of course, a choice as to *who* supplies this particular piece of software:

- QNX Software Systems — you've chosen a standard, supported hardware platform;
- 3rd party — a BIOS or ROM monitor; or
- you — a custom IPL program.

Generally, the simplest development system is one in which you have to do the least amount of work. If we've already done the work, meaning that the board that you're using is a standard, supported hardware platform, there's very little work required from you in this regard; you can instead focus on your software that's going to run on that board.

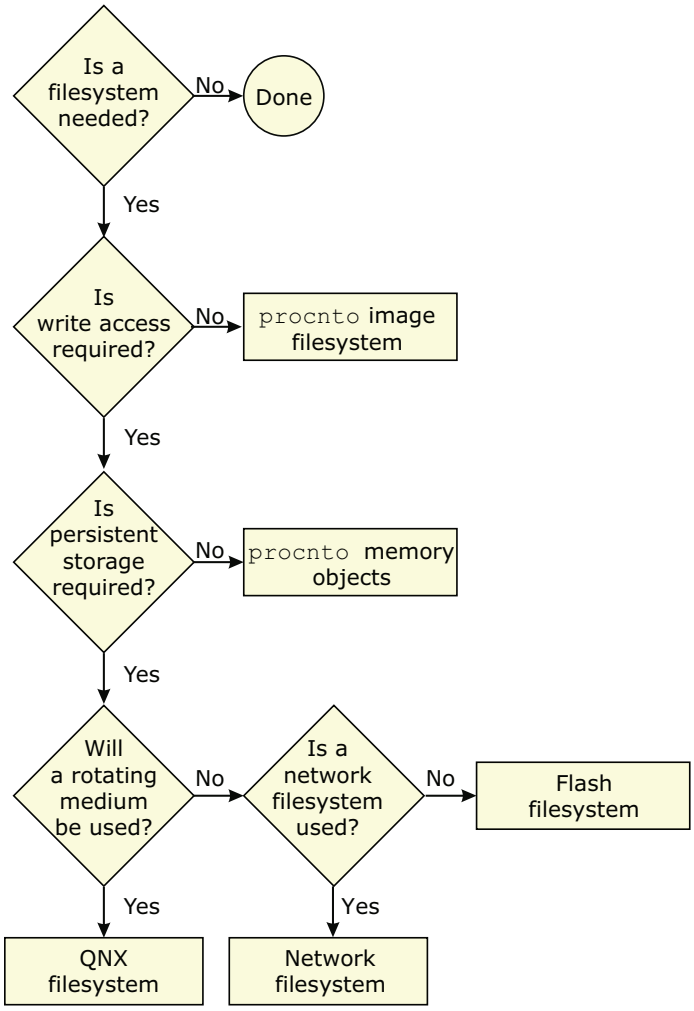
If a 3rd party supplies just the BIOS or ROM monitor, then your responsibilities are increased by having to write the software that starts the operating system. As mentioned earlier, we call this a "warm-start," because the system is already "warmed-up" — various devices are configured and initialized.

If you're supplying a custom IPL, then your responsibilities are further increased by also having to deal with configuration issues for the hardware. This we call a "cold-start," because you are responsible for *everything* to do with initialization and configuration.

Choice of filesystems

Once you've sorted out how the system is going to boot, you may still have additional decisions to make regarding the system's storage capabilities:

- none
- read-only
- read/write nonpersistent
- read/write persistent



You may select as many storage options as you need.

No additional storage required

If you don't require any additional storage (i.e. your system is entirely self-contained and doesn't need to access any other files once it's running), then your work in this regard is done.

Additional read-only storage required

The simplest filesystem scenario is one where read-only access is required. There's no work for you to do — Neutrino provides this functionality as part of the OS itself. Simply place the files that you wish to access/execute directly into the image (see the chapter on Making an OS Image), and the OS will be able to access them.

Additional read/write nonpersistent storage required

If you require write access (perhaps for temporary files, logs, etc.), and the storage doesn't have to be *persistent* in nature (meaning that it doesn't need to survive a reset), then once again the work is done for you.

Neutrino allows the RAM in your system to be used as a RAM-disk, without any additional coding or device drivers. The RAM-disk is implemented via the Process Manager — you simply set up a Process Manager link (using the **ln** command).

For example, to mount the **/tmp** directory as a RAM-disk, execute the following command:

```
ln -Ps /dev/shmem /tmp
```

Or place the following line in your buildfile (we'll talk about buildfiles over the next few chapters):

```
[type=link] /tmp=/dev/shmem
```

This instructs the Process Manager to take requests for any files under **/tmp** and resolve them to the shared memory subsystem. For example, **/tmp/AAA4533.tmp** becomes a request for **/dev/shmem/AAA4533.tmp**.

In order to minimize the size of the RAM filesystem code inside the Process Manager, the shared memory filesystem specifically doesn't include "big filesystem" features such as file locking and directory creation.

If you need a relatively full-featured, POSIX-style filesystem on a RAM disk, use **devf-ram** or the builtin RAM disk via **io-blk** instead.

Additional read/write persistent storage required

If you do require storage that must survive a power failure or processor reset, then you'll need to run an additional driver. We supply these classes of filesystems:

- flash filesystems
- rotating disk filesystems

- network filesystems

All of these filesystems require additional drivers. The Sample Buildfiles appendix in this guide gives detailed examples showing how to set up these filesystem drivers.

Flash filesystems and media

The flash driver can interface to the flash memory devices (boot block and regular) in all combinations of bus widths (8, 16, and 32 bits) and interleave factors (1, 2, and 4).

To find out what flash devices we currently support, please refer to the following sources:

- the **boards** and **mtd-flash** directories under *bsp_working_dir*/**src/hardware/flash**.
- QNX docs (**devf-*** entries in the *Utilities Reference*).
- the QNX Software Systems website (**www.qnx.com**).

Using the source code provided, you may be able to tailor one of our filesystems (e.g. **devf-generic**) to operate on your particular embedded system (if it isn't currently supported).

Rotating media and filesystems

Neutrino currently supports several filesystems, including DOS, Linux, Macintosh HFS and HFS Plus, Windows NT, QNX 4, Power-Safe, Universal Disk Format (UDF), and more. For details, see the **fs-*** entries in the *Utilities Reference*.

Drivers are available for many block-oriented devices. For up-to-date information, see the **devb-*** entries in the *Utilities Reference* as well as the Community area of our website, **www.qnx.com**.

Network media and filesystems

During development, or perhaps in a distributed data-gathering application, you may wish to have a filesystem located on one machine and to be able to access that filesystem from other machines. A network filesystem lets you do this.

In addition to its own transparent distributed processing system (Qnet), QNX Neutrino also supports network filesystems such as CIFS (SMB), NFS 2, and NFS 3.

If possible, you should use **fs-nfs3** instead of **fs-nfs2**.

Drivers are available for the several Ethernet controllers. For details, see the **devn-*** and **devnp-*** entries in the *Utilities Reference* as well as the Community area of our website, **www.qnx.com**.

I/O devices

Ultimately, your Neutrino-based system will need to communicate with the outside world. Here are some of the more common ways to do this:

- serial/parallel port
- network (described above)
- data acquisition/generation
- multimedia

Character I/O devices

For standard serial ports, Neutrino supports several devices (8250 family, Signetics, etc.) For details, see the **`devc-*`** entries in the *Utilities Reference*, as well as the Community area of our website, **`www.qnx.com`**.

Special/custom devices

One design issue you face is whether you can get off-the-shelf drivers for the hardware or whether you'll have to write your own. If it turns out that you need to write your own, then the *Writing a Resource Manager* guide can help you do that.

Getting started

Depending on the ultimate system you'll be creating, you may have a ton of work to do or you may have very little. In any case, we recommend that you start with a supported evaluation board. This approach minimizes the amount of low-level work that you have to do initially, thereby allowing you to focus on your system rather than on implementation details.

Start with an evaluation platform that most closely resembles your target platform — there are many supported evaluation platforms from various vendors.

Once you're comfortable with the development environment and have done a very rudimentary "proof of concept," you can move on to such development efforts as creating your own hardware, writing your own IPL and startup code, writing drivers for your hardware, and so on.

Your proof of concept should address such issues as:

- How much memory will be required?
- How fast a CPU will be required?
- Can standard off-the-shelf hardware do the job?

Once these are addressed, you can then decide on your plan of attack.

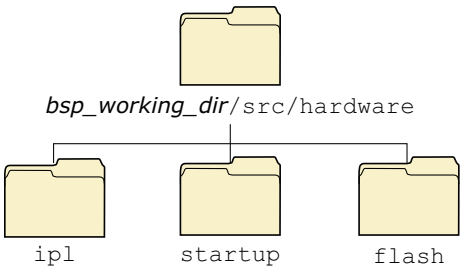
Hardware design

There are a number of ways of designing your hardware. We've seen many boards come in from the field and have documented some of our experiences with them in the System Design Considerations appendix in this book. You may be able to realize certain savings (in both cost and time) by reading that appendix first.

Customizing the software

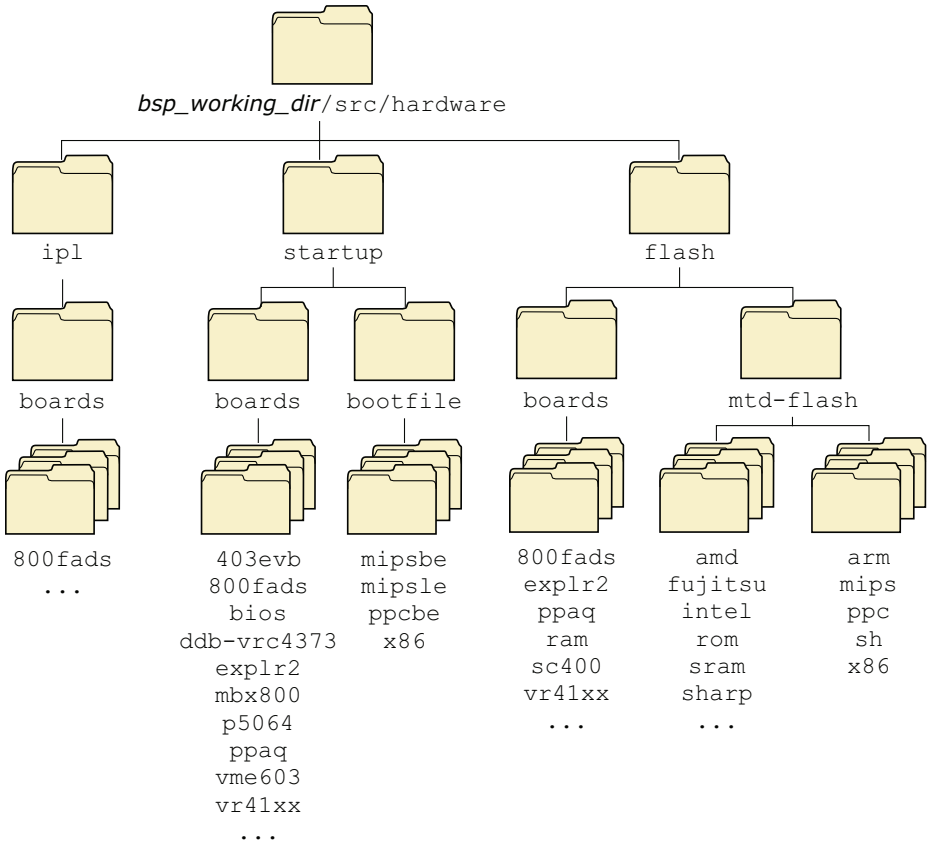
Ideally, the system you're designing will look identical to a supported evaluation platform. In reality, this isn't always the case, so you'll need to customize some of the components in that system.

We've provided the source code to a large number of the "customizable" pieces of the OS. This diagram gives you the high-level view of the directory structure for the source tree we ship:



The three main branches of the Neutrino source tree.

As you can see, we've divided the source tree into three major branches: **`ipl`**, **`startup`**, and **`flash`**. Each branch consists of further subdirectories:



The complete Neutrino source tree.

Customizing the source

The following table relates the source tree branches to the individual chapters in this book:

Source tree branch	Relevant chapter
`ipl`	Customizing IPL Programs
`startup`	Customizing Image Startup Programs
`flash`	Customizing the Flash Filesystem

For detailed information on the format of the **`Makefile`** present in these directories, see Conventions for Recursive Makefiles and Directories in the QNX Neutrino *Programmer's Guide*.

Chapter 2

Working with a BSP

In this chapter. . .

BSP Overview

Once you've installed the QNX Software Development Platform, you can download processor-specific Board Support Packages (BSPs) from our website, **`http://www.qnx.com/`**. These BSPs are designed to help you get Neutrino running on certain platforms.

A BSP typically includes the following:

- IPL
- startup
- default buildfile
- networking support
- board-specific device drivers, system managers, utilities, etc.

The BSP is contained in an archive named after the industry-recognized name of the board and/or reference platform that the BSP supports. BSP packages are available for QNX Neutrino, Windows, or Linux hosts.

The BSP components are provided in source code form, unless there are restrictions on the source code, in which case the component is provided only in binary form. BSPs are provided in a **`zip`** archive. The same archive applies to all hosts.

The QNX community website, Foundry27, has more information about BSPs:

- For information about using BSPs from earlier releases, see:

  `http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/PRE640BSP_migrationDoc`

- For information about packaging a BSP, see:

  `http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/Packaging_BSP`

You can also check out BSPs from a Subversion repository on Foundry27.

To use a BSP, you must either unzip the archive and build it on the command line, or import it into the IDE.

Using BSPs in the IDE

Before working with a BSP in the IDE, you must first import it. When you import the BSP source, the IDE creates a System Builder project.

To import the BSP source code:

1. Select **File→Import**.
2. Expand the **QNX** folder.

3. Select **QNX Board Support Package** from the list. Click **Next**.
4. In the **Select the package to import** dialog, click **Select Package**, and then choose the BSP archive using the file browser.
5. Choose the BSP you want. You'll see a description of it.
6. Click **Next**.
7. Uncheck the entries you don't want imported. (By default all the entries are selected.)
8. Click **Next**.
9. Select a working set. Default names are provided for the **Working Set Name** and the **Project Name Prefix** that you can override if you choose.
10. Click **Finish**. All the projects will be created and the source brought from the archive. You'll then be asked if you want to build all the projects you've imported.

    If you answer Yes, the IDE will start the build process. If you decide to build at a later time, you can do a **Rebuild All** from the main **Project** menu when you're ready to build.

    When you import a QNX BSP, the IDE opens the QNX BSP Perspective. This perspective combines the minimum elements from the C\C++ Development Perspective and the System Builder Perspective.

For more information, see the IDE *User's Guide* in your documentation set. (Within the IDE itself, go to: **Help→Help Contents→QNX Documentation Roadmap**).

Using BSPs on the command line

If you aren't using the IDE and you want to manually install a BSP archive, we recommend that you create a default directory with the same name as your BSP and unzip the archive from there:

1. Change the directory to where you want to extract the BSP (e.g. **`/home/joe`**). The archive will extract to the current directory, so you should create a directory specifically for your BSP.

   For example:

   `mkdir /home/joe/`*`bspname`*

2. In the directory you've just created, extract the BSP:

   `cd /home/joe/`*`bspname`*
   `unzip` *`bspname`***`.zip`**

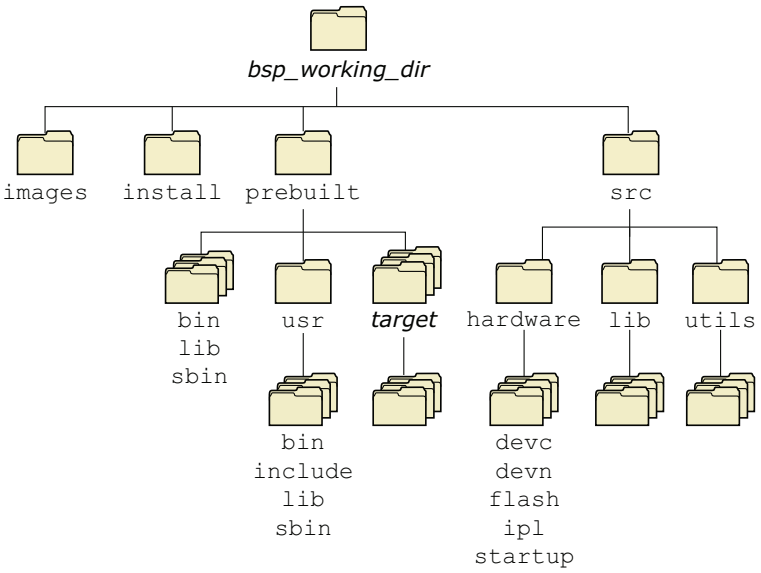
See Foundry27 for instructions on how to get a BSP from Subversion.

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

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

Structure of a BSP

After you unzip a BSP archive, the resulting directory structure looks something like this:



BSP directory structure.

In our documentation, we refer to the directory where you've installed a BSP (e.g. **`/home/myID/my_BSPs/integrator`**) as the *bsp_working_dir*. This directory includes the following subdirectories:

- **`src`**
- **`prebuilt`**
- **`install`**
- **`images`**

The **images** subdirectory is where the resultant boot images are placed. It contains (as a minimum) the **Makefile** needed to build the image(s). Other files that could reside in this directory include:

- custom buildfiles (for flash, etc.)
- EFS buildfiles
- IPL build scripts

prebuilt subdirectory

The **prebuilt** subdirectory contains prebuilt binaries, and header files that are shipped with the BSP.

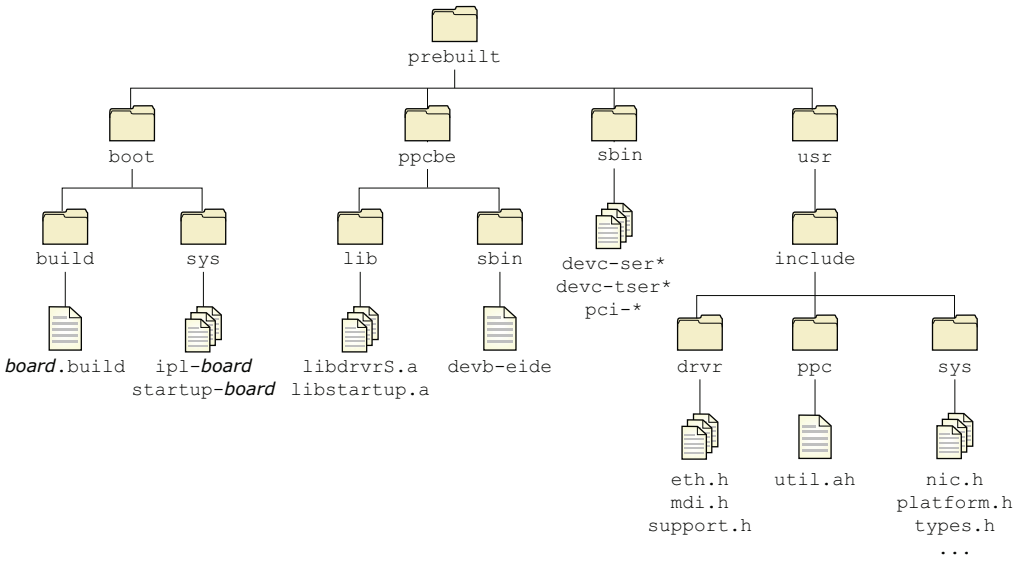
Before the BSP is built, all of the files from the **prebuilt** directory are copied into the **install** directory, maintaining the path structure.

In order to handle dependencies, the libraries, headers, and other files found in the **./prebuilt** directory need to be copied correctly to your **./install** directory. To do this, you'll need to run **make** at the *bsp_working_dir* directory level.

The "root" of the **prebuilt** directory requires the same structure as the system root. The target-specific and **usr** directories mirror the structure of **/**.

All processor-specific binaries are located under the directory named for that processor type.

For example, the **prebuilt** directory might look like this:



A sample **prebuilt** *directory.*

`install` subdirectory

The **`install`** directory gets populated at the beginning of the BSP build process. All the files in the **`prebuilt`** directory are copied, then all generated binaries are installed here as they're compiled. The files stored in the **`install`** directory are taken first when **`mkifs`** executes.

Before you make any components for your particular board, you must first make the BSP sources *at the top level*:

```
cd bsp_working_dir
make
```

This builds everything under **`./src`** and sets up the **`./install`** and **`./images`** subdirectories correctly.

After this initial build is complete, you can build any of the source files individually.

If you change a library or header, be sure to run **`make install`** to rebuild the source and copy the changes to your **`./install`** directory.

`src` subdirectory

The BSP-specific source code is stored in this directory. Refer to the BSP release notes to find the location of the source code for a specific driver.

The **`hardware`** directory contains separate directories for character, flash, and network drivers, IPL, startup code, and so on, depending on the BSP.

The **`src`** directory contains one or more master buildfiles, typically **`src/hardware/startup/boards/`***board*`/build`**. During **`make install`** the build files are copied to **`install/`***target*`/boot/build/`***board*`.build`**. After the root Makefile will make a link to, or make a copy of these files in the **`images`** subdirectory. Care is required to modify the correct buildfile and to avoid losing changes to a buildfile.

The **`lib`** directory contains separate directories for libraries that are required by driver and other utilities that are included with the BSP.

Some drivers, such as the network drivers or USB host controller drivers, are implemented as shared objects, but the source code for them is located under the **`hardware`** directory.

The **`utils`** directory contains separate directories for minor utilities that are required on the board. Some hardware-specific utilities can also be found in **`hardware/support`**.

The **`services`** directory contains separate directories for additional services that aren't included in the base installation.

Building source from the command line

When you build a BSP from the source code, you may occasionally observe warnings from some of the tools used to generate the BSP, such as:

- **`objcopy: Warning: Output file cannot represent architecture UNKNOWN!`**
- **`ntosh-ld: Warning: could not find any targets that match endian ness requirement`**

These warnings result when information that's contained in one particular file format (endian ness, CPU architecture, etc.) can't be retained when converting that file to a different format, or when the originating file format doesn't contain information that the tool doing the conversion expects. These warnings are normal and expected, and are no cause for concern.

In order to build a BSP from the command line, you must go to the root directory for the BSP.

Use the **`make`** command to build the source code. The **`Makefile`** defines the following targets:

`all` Invokes the **`install`**, **`links`**, and **`images`** targets.

`prebuilt` This recursively copies the **`prebuilt`** directory's contents to the **`install`** directory.

`install` Invokes the **`prebuilt`** target, and then performs the following in the **`src`** directory:

- **`make hinstall`** to copy all public headers from **`src`** into the **`install`** directory.
- **`make install`** to build all binaries in **`src`** and copy the results into the **`install`** directory. This target also copies the buildfile from **`src/hardware/startup/boards/`***board*`/build`** and renames it *board*`.build`.

`links` Creates a symbolic link (a copy on Windows) from **`install/`***cpu*`/boot/build/`***board*`.build`** to **`images/`***board*`.build`**.

`images` Changes to the **`images`** directory and runs the **`Makefile`** there. This **`Makefile`** creates an IFS file based on the buildfile linked in during the **`make links`** target. Any extra work required (e.g. IPL padding, conversion to an alternate format) is also handled from within this **`Makefile`**.

If you don't specify a target, **`make`** invokes the **`all`** target.

We recommend that you use **make** to build the OS image. If you use **mkifs** directly, you need to use the **-r** option to specify where to find the binaries. For more information, see the entry for **mkifs** in the *Utilities Reference*.

Supporting additional devices

All boards have some devices, whether they're input, serial, flash, or PCI. Every BSP includes a buildfile that you can use to generate an OS image that will run on the board it was written for. The buildfile is in the *bsp_working_dir*/**src/hardware/startup/boards/***board* directory.

A BSP's buildfile contains the commands — possibly commented out — for starting the drivers associated with the devices. You will need to edit the buildfile to modify or uncomment these commands. If you uncomment a command, make sure you uncomment the lines that add any required binaries to the image.

For more information, see the documentation for each BSP, as well as the buildfile itself; for general information about buildfiles, see the entry for **mkifs** in the *Utilities Reference*.

Once you've modified the buildfile, follow the instructions given earlier in this chapter for building an OS image.

Transferring an OS image onto your board

Once you've built an OS image, you'll need to transfer it to your board.

The IDE lets you communicate with your target and download your OS image using either a serial connection, or a network connection using the Trivial File Transfer Protocol (TFTP). If your board doesn't have a ROM monitor, you probably can't use the download services in the IDE; you'll have to get the image onto the board some other way (e.g. JTAG).

Transferring an OS image

There are several ways to transfer an OS image:

To:	Use the:
Load an image from your network (e.g. TFTP)	Network
Load an image serially (e.g. COM1, COM2)	ROM monitor
Burn both the IPL and the OS image into the flash boot ROM, then boot entirely from flash	IPL and OS

continued...

To:	Use the:
Burn an IPL (Initial Program Loader) into the flash boot ROM, then load the OS image serially	IPL and boot ROM
Generate a flash filesystem, and then place various files and utilities within it	Flash filesystem

The method you use to transfer an OS image depends on what comes with the board. The BSP contains information describing the method that you can use for each particular board. Each board will have all or some of these options for you to use.

To load an image serially:

1. Connect your target and host machine with a serial cable. Ensure that both machines properly recognize the connection.
2. Specify the device (e.g.COM1) and the communications settings (e.g. the baud rate, parity, data bits, stop bits, and flow control) to match your target machine's capabilities. You can now interact with your target by typing in the view.

To transfer a file using the **Serial Terminal** view:

1. Using either the serial terminal view or another method (outside the IDE), configure your target so that it's ready to receive an image.
2. In the serial terminal view, click **Send File**.
3. In the **Select File to Send** dialog, enter the name or your file (or click **Browse**).
4. Select a protocol (e.g. **`sendnto`**).
5. Click OK. The Builder transmits your file over the serial connection.

Working with a flash filesystem

The flash filesystem drivers implement a POSIX-like filesystem on NOR flash memory devices. The flash filesystem drivers are standalone executables that contain both the flash filesystem code and the flash device code. There are versions of the flash filesystem driver for different embedded systems hardware as well as PCMCIA memory cards.

The naming convention for the drivers is **`devf-`***system*, where *system* describes the embedded system. For example, the **`devf-800fads`** driver is for the 800FADS PowerPC evaluation board.

To find out what flash devices we currently support, please refer to the following sources:

- the **`boards`** and **`mtd-flash`** directories under *bsp_working_dir***`/src/hardware/flash`**
- QNX Neutrino OS docs (**`devf-*`** entries in *Utilities Reference*

- the QNX Software Systems website (**www.qnx.com**)

The flash filesystem drivers support one or more logical flash drives. Each logical drive is called a *socket*, which consists of a contiguous and homogeneous region of flash memory. For example, in a system containing two different types of flash device at different addresses, where one flash device is used for the boot image and the other for the flash filesystem, each flash device would appear in a different socket.

Each socket may be divided into one or more partitions. Two types of partitions are supported:

- raw partitions
- flash filesystem partitions

Raw partitions

A raw partition in the socket is any partition that doesn't contain a flash filesystem. The flash filesystem driver doesn't recognize any filesystem types other than the flash filesystem. A raw partition may contain an image filesystem or some application-specific data.

The flash filesystem uses a raw mountpoint to provide access to any partitions on the flash that aren't flash filesystem partitions. Note that the flash filesystem partitions are available as raw partitions as well.

Flash filesystem partitions

A flash filesystem partition contains the POSIX-like flash filesystem, which uses a QNX-proprietary format to store the filesystem data on the flash devices. This format isn't compatible with either the Microsoft FFS2 or PCMCIA FTL specification.

The flash filesystem allows files and directories to be freely created and deleted. It recovers space from deleted files using a reclaim mechanism similar to garbage collection.

The flash filesystem supports all the standard POSIX utilities such as **ls**, **mkdir**, **rm**, **ln**, **mv**, and **cp**. There are also some QNX Neutrino utilities for managing the flash filesystem:

flashctl	Erase, format, and mount flash partitions.
deflate	Compress files for flash filesystems.
mkefs	Create flash filesystem image files.

The flash filesystem supports all the standard POSIX I/O functions such as *open()*, *close()*, *read()*, and *write()*. Special functions such as erasing are supported using the *devctl()* function.

Flash filesystem source

Each BSP contains the binary and the source code for the appropriate flash filesystem driver, but the QNX Software Development Platform contains the associated header files and libraries.

Typing **`make`** in the *bsp_working_dir* generates the flash filesystem binary. Normally, you won't need to remake the flash filesystem driver unless you've changed the size or configuration of the flash on the board — this can include the number of parts, size of parts, type of parts, interleave, etc.

CAUTION: When an IPL/IFS (image filesystem) image is combined, you'll need to offset the beginning of the flash filesystem by at least the size of the IPL and IFS. For example, if the combined IPL/IFS image is loaded at offset 0 on the flash, to avoid overwriting the IPL and IFS, the flash filesystem must begin at an offset of the IPL/IFS image size +1. If it doesn't begin at an offset of the IPL/IFS image size +1, you'll need to create a partition.

How do I create a partition?

Regardless of which BSP you're working with, the procedure requires that you:

1. Start the flash filesystem driver.
2. Erase the entire flash.
3. Format the partition.
4. Slay the flash filesystem driver.
5. Restart the flash filesystem driver.

The following example applies specifically to the Renesas Biscayne board, which can be booted from DMON or flash.

1. To boot from DMON, enter the following command to start the flash filesystem driver:

   ```
   devf-generic -s0xe8000000,32M &
   ```

2. To boot from flash, enter the following command to start the flash system driver:

   ```
   devf-generic -s0x0,32M
   ```

   You should now see an **`fs0p0`** entry under **`/dev`**.

3. To prepare the area for the partition, you must erase the entire flash. Enter the following command:

   ```
   flashctl -p/dev/fs0 -ev
   ```

4. To format the partition, enter the following command:

   ```
   flashctl -p/dev/fs0p0 -f
   ```

5 Now slay the flash filesystem driver:

```
slay devf-generic
```

6 Finally, restart the driver:

```
devf-generic &
```

You should now see the following entries:

Entry	Description
`/dev/fs0p0`	OS image (32 MB)
`/dev/fs0p1`	Flash filesystem partition (32 MB)

Testing Neutrino on your board

You can test Neutrino simply by executing any shell builtin command or any command residing within the OS image. For example, type:

```
ls
```

You'll see a directory listing, since the **ls** command has been provided in the default system image.

Getting Photon on your board

For instructions on adding the Photon microGUI to your embedded system, see the documentation for the particular BSP; the buildfile could include the specific commands (commented out) that you need to run. For even more details, see the "Photon in Embedded Systems" appendix in the Photon *Programmer's Guide*.

Where do I go from here?

Now that you have a better understanding of how BSPs work in an embedded system, you'll want to start working on your applications. The following table contains references to the QNX documentation that may help you find the information you'll need to get going.

For information on:	Go to:
Writing "hello world"	The section "A simple example" in the chapter Compiling and Debugging in the Neutrino *Programmer's Guide*, or the *IDE User's Guide*.

continued...

For information on:	Go to:
Debugging your programs	The section “Debugging” in the chapter Compiling and Debugging in the Neutrino *Programmer’s Guide*.
Setting up NFS	The section “Complete example — TCP/IP with network filesystem” in the appendix Sample Buildfiles in this manual. See also the **`fs-nfs3`** utility page in the *Utilities Reference*.
Setting up an Ethernet driver	The section “Complete example — TCP/IP with network filesystem” in the appendix Sample Buildfiles in this manual. See also the various network drivers (**`devn*`**, **`devnp-*`**) in the *Utilities Reference*.
Writing device drivers and/or resource managers	*Writing a Resource Manager*

If you need more information, see these chapters in this guide:

For more information on:	Go to:
Building flash filesystems	Customizing the Flash Filesystem
IPL	Writing an IPL program
Startup	Customizing Image Startup Programs

Filename conventions

In QNX Neutrino BSPs, we use the following conventions for naming files:

Part of filename	Description	Example
`.bin`	Suffix for binary format file	**`ifs-artesyn.bin`**
`.build`	Suffix for buildfile	**`sandpoint.build`**
`efs-`	Prefix for QNX Embedded Filesystem file; generated by **`mkefs`**	**`efs-sengine.srec`**
`.elf`	Suffix for ELF (Executable and Linking Format) file	**`ipl-ifs-mbx800.elf`**

continued...

Part of filename	Description	Example
`ifs-`	Prefix for QNX Image Filesystem file; generated by `mkifs`	`ifs-800fads.elf`
`ipl-`	Prefix for IPL (Initial Program Loader) file	`ipl-eagle.srec`
`.openbios`	Suffix for OpenBIOS format file	`ifs-walnut.openbios`
`.prepboot`	Suffix for Motorola PRePboot format file	`ifs-prpmc800.prepboot`
`.srec`	Suffix for S-record format file	`ifs-malta.srec`

Chapter 3

Making an OS Image

In this chapter. . .

Making an OS image involves a number of steps, depending on the hardware and configuration of your target system.

In this chapter, we'll take a look at the steps necessary to build an OS image. Then we'll examine the steps required to get that image to the target, whether it involves creating a boot disk/floppy, a network boot, or burning the image into an EPROM or flash device. We'll also discuss how to put together some sample systems to show you how to use the various drivers and resource managers that we supply.

For more information on using the various utilities described in this chapter, see the *Utilities Reference*.

Images, images, images

In the embedded Neutrino world, an "*image*" can mean any of the following:

Image type	Description	Created by:
OS image	A bootable or nonbootable structure that contains files	**mkifs**
Flash filesystem image	A structure that can be used in a read-only, read/write, or read/write/reclaim flash filesystem	**mkefs**
Embedded transaction filesystem image	A binary image file containing the ETFS as a sequence of transactions	**mketfs**

What is an OS image?

When you've created your executables (programs) that you want your embedded system to run, you need to place them somewhere where they can be loaded from. An OS image is simply a file that contains the OS, your executables, and any data files that might be related to your programs. Actually, you can think of the image as a small "filesystem" — it has a directory structure and some files in it.

An image can be *bootable* or *nonbootable*. A bootable image is one that contains the startup code that the IPL can transfer control to (see the chapter on customizing IPL programs in this book). Generally, a small embedded system will have only the one (bootable) OS image.

A nonbootable image is usually provided for systems where a separate, configuration-dependent setup may be required. Think of it as a second "filesystem" that has some additional files in it (we'll discuss this in more depth later). Since it's nonbootable, this image will typically *not* contain the OS, startup file, etc.

The OS image as a filesystem

As previously mentioned, the OS image can be thought of as a filesystem. In fact, the image contains a small directory structure that tells **procnto** the names and positions of the files contained within it; the image also contains the files themselves. When the embedded system is running, the image can be accessed just like any other read-only filesystem:

```
# cd /proc/boot
# ls
.script     ping        cat       data1        pidin

ksh         ls          ftp       procnto      devc-ser8250-abc123
# cat data1
This is a data file, called data1, contained in the image.
Note that this is a convenient way of associating data
files with your programs.
```

The above example actually demonstrates two aspects of having the OS image function as a filesystem. When we issued the **ls** command, the OS loaded **ls** from the image filesystem (pathname **/proc/boot/ls**). Then, when we issued the **cat** command, the OS loaded **cat** from the image filesystem as well, and opened the file **data1**.

Let's now take a look at how we configure the image to contain files.

Configuring an OS image

The OS image is created by a program called **mkifs** (*mk*e *i*mage *f*ilesystem), which accepts information from two main sources: its command line and a *buildfile*.

For more information, see **mkifs** in the *Utilities Reference*.

A simple buildfile

Let's look at a very simple buildfile, the one that generated the OS image used in the example above:

```
#   A simple "ls", "ping", and shell.
#   This file is "shell.bld"

[virtual=armle,srec] .bootstrap = {
    startup-abc123
    PATH=/proc/boot procnto -vv
}
[+script] .script = {
    procmgr_symlink ../../proc/boot/libc.so.3 /usr/lib/ldqnx.so.2

    devc-ser8250-abc123 -F -e -c14745600  -b115200 0xc8000000 ^2,15 &
    reopen

    display_msg Serial Driver Started
}

[type=link] /dev/console=/dev/ser1
[type=link] /tmp=/dev/shmem
libc.so.2
libc.so
```

```
[data=copy]
devc-ser8250-abc123
ksh
ls
cat
data1
ping
ftp
pidin
```

In a buildfile, a pound sign (**#**) indicates a comment; anything between it and the end of the line is ignored. Make sure there's a space between a buildfile command and the pound sign.

This buildfile consists of these sections:

- a *bootfile* — starting with **[virtual=armle,srec]**
- a script — starting with **[+script]**
- a list of links and files to include in the image — starting with **[type=link] /dev/console=/dev/ser1**

Inline files

Although the three sections in the buildfile above seem to be distinct, in reality all three are similar in that they're lists of files.

Notice also how the buildfile itself is structured:

optional_attributes filename optional_contents

For example, the line:

```
[virtual=armle,srec] .bootstrap = {
```

has an attribute of **[virtual=armle,srec]** and a filename of **.bootstrap**. The *optional_contents* part of the line is what we call an *inline file*; instead of getting the contents of this file from the host machine, **mkifs** gets them from the buildfile itself, enclosed by braces. The contents of the inline file can't be on the same line as the opening or closing brace.

Let's examine these elements in some detail.

The bootstrap file

The first section of the bootfile (starting with **[virtual=armle,srec]**) specifies that a virtual address system is being built. The CPU type appears next; "**armle**" indicates a little-endian ARM processor. Then after the comma comes the name of the bootfile (**srec**).

The rest of the line specifies an inline file (as indicated by the open brace) named "**.bootstrap**", which consists of the following:

```
startup-abc123
PATH=/proc/boot procnto -vv
```

If you set the value of **PATH** in the bootstrap file, **procnto** sets the _CS_PATH configuration string. Similarily, if you set **LD_LIBRARY_PATH**, **procnto** sets the _CS_LIBPATH configuration string. It doesn't pass these environment variables on to the script, but you *can* set environment variables in the script itself.

You can bind in optional modules to **procnto** by using the **[module=...]** attribute. For example, to bind in the adaptive partitioning scheduler, change the **procnto** line to this:

```
[module=aps] PATH=/proc/boot procnto -vv
```

- Optional modules to **procnto** were introduced in the QNX Neutrino Core OS 6.3.2.
- For more information about the adaptive partitioning scheduler, see the Adaptive Partitioning *User's Guide*.

The actual name of the bootstrap file is irrelevant. However, nowhere else in the buildfile did we specify the bootstrap or script files — they're included automatically when specified by a **[virtual]** or **[physical]** attribute.

The "**virtual**" attribute (and its sibling the "**physical**" attribute) specifies the target processor (in our example, the **armle** part) and the bootfile (the **srec** part), a very small amount of code between the IPL and startup programs. The target processor is put into the environment variable **$PROCESSOR** and is used during pathname expansion. You can omit the target processor specification, in which case it defaults to the same as the host processor. For example:

```
[virtual=bios] .bootstrap = {
...
```

would assume an ARM target if you're on an ARM host system.

Both examples find a file called **$PROCESSOR/sys/bios.boot** (the **.boot** part is added automatically by **mkifs**), and process it for configuration information.

Compressing the image

While we're looking at the bootstrap specification, it's worth mentioning that you can apply the **+compress** attribute to compress the entire image. The image is automatically uncompressed before being started. Here's what the first line would look like:

```
[virtual=armle,srec +compress] .bootstrap = {
```

The script file

The second section of the buildfile starts with the **[+script]** attribute — this tells **mkifs** that the specified file is a *script file*, a sequence of commands that you want **procnto** to execute when it's completed its own startup.

Script files look just like regular shell scripts, except that:

- special modifiers can be placed *before* the actual commands to run
- some commands are builtin
- the script file's contents are parsed by **mkifs** before being placed into the image

In order to run a command, its executable must be available when the script is executed. You can add the executable to the image or get it from a filesystem that's started before the executable is required. The latter approach results in a smaller image.

In this case, the script file is an inline file (again indicated by the open brace). The file (which happens to be called "**.script**") contains the following:

```
procmgr_symlink ../../proc/boot/libc.so.3 /usr/lib/ldqnx.so.2

devc-ser8250-abc123 -F -e -c14745600  -b115200 0xc8000000 ^2,15 &
reopen

display_msg Serial Driver Started
```

This script file begins by creating a symbolic link to **../../proc/boot/libc.so.3** called **/usr/lib/ldqnx.so.2**. Next the script starts a serial driver (the fictional **devc-ser8250-abc123**) in edited mode with hardware flow control disabled at a baud rate of 115200 bps at a particular physical memory address. The script then does a **reopen** to redirect standard input, output, and error. The last line simply displays a message.

As mentioned above, the bootstrap file can set the _CS_PATH and _CS_LIBPATH configuration strings. You can set **PATH**, **LD_LIBRARY_PATH**, and other environment variables if the programs in your script need them.

CAUTION: If you specify an ampersand (**&**) after the command line, the program runs in the background, and Neutrino doesn't wait for the program to finish before continuing with the next line in the script.

If you don't specify the ampersand, and the program doesn't exit, then the rest of the script is never executed, and the system doesn't become fully operational. In particular, **procnto** doesn't reap zombies that get reparented to it, resulting in a system that accumulates zombie processes, all parented to **procnto**, that won't go away until you reboot.

Bound multiprocessing attributes

You can specify which CPU to bind processes to when launching processes from the startup script through the **[CPU=]** modifier.

The **[CPU=]** is used as any other modifier, and specifies the CPU on which to launch the following process (or, if the attribute is used alone on a line without a command, sets the default CPU for all following processes). Specify the CPU as a zero-based processor number:

```
[cpu=0] my_program
```

A value of ***** allows the processes to run on all processors:

```
[cpu=*] my_program
```

At boot time, if there isn't a processor with the given index, a warning message is displayed, and the command is launched without any runmask restriction.

Due to a limitation in the boot image records, this syntax allows only the specification of a single CPU and not a more generic runmask. Use the **on** utility to spawn a process within a fully specified runmask.

The script file on the target

The script file stored on the target isn't the same as the original specification of the script file within the buildfile. That's because a script file is "special" — **mkifs** parses the text commands in the script file and stores only the parsed output on the target, not the original ASCII text. The reason we did this was to minimize the work that the process manager has to do at runtime when it starts up and processes the script file — we didn't want to have to include a complete shell interpreter within the process manager!

Plain ordinary lists of files

Let's return to our example. Notice the "list of files" (i.e. from "**[type=link] /dev/console=/dev/ser1**" to "**pidin**").

Including files from different places

In the example above, we specified that the files at the end were to be part of the image, and **mkifs** somehow magically found them. Actually, it's not magic — **mkifs** simply looked for the environment variable **MKIFS_PATH**. This environment variable contains a list of places to look for the files specified in the buildfile. If the environment variable doesn't exist, then the following are searched in this order:

1. current working directory *if* the filename contains a slash (but doesn't start with one).
2. **${**QNX_TARGET**}/${**PROCESSOR**}/sbin**
3. **${**QNX_TARGET**}/${**PROCESSOR**}/usr/sbin**

4 `${QNX_TARGET}/${PROCESSOR}/boot/sys`

5 `${QNX_TARGET}/${PROCESSOR}/bin`

6 `${QNX_TARGET}/${PROCESSOR}/usr/bin`

7 `${QNX_TARGET}/${PROCESSOR}/lib`

8 `${QNX_TARGET}/${PROCESSOR}/lib/dll`

9 `${QNX_TARGET}/${PROCESSOR}/usr/lib`

10 `${QNX_TARGET}/${PROCESSOR}/usr/photon/bin`

(The `${PROCESSOR}` component is replaced with the name of the CPU, e.g. **arm**.)

Since none of the filenames that we used in our example *starts* with the "**/**" character, we're telling **mkifs** that it should search for files (on the host) within the path list specified by the **MKIFS_PATH** environment variable as described above. Regardless of where the files came from on the host, in our example they'll all be placed on the target under the **/proc/boot** directory (there are a few subtleties with this, which we'll come back to).

For our example, **devc-con** will appear on the target as the file **/proc/boot/devc-con**, even though it may have come from the host as `${QNX_TARGET}/armle/sbin/devc-con`.

To include files from locations other than those specified in the **MKIFS_PATH** environment variable, you have a number of options:

- Change the **MKIFS_PATH** environment variable (use the shell command **export MKIFS_PATH=***newpath* on the host).
- Modify the search path with the **[search=]** attribute.
- Specify the pathname explicitly (i.e. with a leading "**/**" character).
- Create the contents of the file in line.

Modifying the search path

By specifying the **[search=***newpath***]** attribute, we can cause **mkifs** to look in places other than what the environment variable **MKIFS_PATH** specifies. The *newpath* component is a colon-separated list of pathnames and can include environment variable expansion. For example, to augment the existing **MKIFS_PATH** pathname to also include the directory **/mystuff**, you would specify:

```
[search=${MKIFS_PATH}:/mystuff]
```

Specifying the pathname explicitly

Let's assume that one of the files used in the example is actually stored on your development system as **/release/data1**. If you simply put **/release/data1** in the buildfile, **mkifs** would include the file in the image, but would call it **/proc/boot/data1** on the target system, instead of **/release/data1**.

Sometimes this is exactly what you want. But at other times you may want to specify the *exact pathname* on the target (i.e. you may wish to override the prefix of **/proc/boot**). For example, specifying **/etc/passwd** would place the host filesystem's **/etc/passwd** file in the target's pathname space as **/proc/boot/passwd** — most likely not what you intended. To get around this, you could specify:

```
/etc/passwd = /etc/passwd
```

This tells **mkifs** that the file **/etc/passwd** on the host should be stored as **/etc/passwd** on the target.

On the other hand, you may in fact want a different source file (let's say **/home/joe/embedded/passwd**) to be the password file for the embedded system. In that case, you would specify:

```
/etc/passwd = /home/joe/embedded/passwd
```

Creating the contents of the file in line

For our tiny **data1** file, we could just as easily have included it in line — that is to say, we could have specified its contents directly in the buildfile itself, without the need to have a real **data1** file reside somewhere on the host's filesystem. To include the contents in line, we would have specified:

```
data1 = {
This is a data file, called data1, contained in the image.
Note that this is a convenient way of associating data
files with your programs.
}
```

A few notes. If your inline file contains the closing brace ("**}**"), then you must escape that closing brace with a backslash ("\"). This also means that all backslashes must be escaped as well. To have an inline file that contains the following:

```
This includes a {, a }, and a \ character.
```

you would have to specify this file (let's call it **data2**) as follows:

```
data2 = {
This includes a {, a \}, and a \\ character.
}
```

Note that since we didn't want the **data2** file to contain leading spaces, we didn't supply any in the inline definition. The following, while perhaps "better looking," would be incorrect:

```
# This is wrong, because it includes leading spaces!
data2 = {
    This includes a {, a \}, and a \\ character.
}
```

If the filename that you're specifying has "weird" characters in it, then you must quote the name with double quote characters (**"**). For example, to create a file called **I "think" so** (note the spaces and quotation marks), you would have to specify it as follows:

```
"I \"think\" so" = ...
```

But naming files like this is discouraged, since the filenames are somewhat awkward to type from a command line (not to mention that they look goofy).

Specifying file ownership and permissions

The files that we included (in the example above) had the owner, group ID, and permissions fields set to whatever they were set to on the host filesystem they came from. The inline files (**data1** and **data2**) got the user ID and group ID fields from the user who ran the **mkifs** program. The permissions are set according to the user's **umask**.

If we wanted to explicitly set these fields on particular files within the buildfile, we would prefix the filenames with an attribute:

```
[uid=0 gid=0 perms=0666] file1
[uid=5 gid=1 perms=a+xr] file2
```

This marks the first file (**file1**) as being owned by **root** (the user ID **0**), group zero, and readable and writable by all (the mode of octal **666**). The second file (**file2**) is marked as being owned by user ID **5**, group ID **1**, and executable and readable by all (the **a+xr** permissions).

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

Notice how when we combine attributes, we place all of the attributes within one open-square/close-square set. The following is incorrect:

```
# Wrong way to do it!
[uid=0] [gid=0] [perms=0666] file1
```

If we wanted to set these fields for a bunch of files, the easiest way to do that would be to specify the **uid**, **gid**, and **perms** attributes on a single line, followed by the list of files:

```
[uid=5 gid=1 perms=0666]
file1
file2
file3
file4
```

which is equivalent to:

```
[uid=5 gid=1 perms=0666] file1
[uid=5 gid=1 perms=0666] file2
[uid=5 gid=1 perms=0666] file3
[uid=5 gid=1 perms=0666] file4
```

Including a whole whack of files

If we wanted to include a large number of files, perhaps from a preconfigured directory, we would simply specify the name of the directory instead of the individual filenames. For example, if we had a directory called **/release_1.0**, and we wanted all the files under that directory to be included in the image, our buildfile would have the line:

```
/release_1.0
```

This would put all the files that reside under **/release_1.0** into **/proc/boot** on the target. If there were subdirectories under **/release_1.0**, then they too would be created under **/proc/boot**, and all the files in those subdirectories would also be included in the target.

Again, this may or may not be what you intend. If you really want the **/release_1.0** files to be placed under **/**, you would specify:

```
/=/release_1.0
```

This tells **mkifs** that it should grab everything from the **/release_1.0** directory and put it into a directory called **/**. As another example, if we wanted everything in the host's **/release_1.0** directory to live under **/product** on the target, we would specify:

```
/product=/release_1.0
```

Generating the image

To generate the image file from our sample buildfile, you could execute the command:

```
mkifs shell.bld shell.ifs
```

This tells **mkifs** to use the buildfile **shell.bld** to create the image file **shell.ifs**.

You can also specify command-line options to **mkifs**. Since these command-line options are interpreted *before* the actual buildfile, you can add lines before the buildfile. You would do this if you wanted to use a makefile to change the defaults of a generic buildfile.

The following sample changes the address at which the image starts to 64 KB (hex **0x10000**):

```
mkifs -l "[image=0x10000]" buildfile image
```

For more information, see **mkifs** in the *Utilities Reference*.

Listing the contents of an image

If you'd like to see the contents of an image, you can use the **dumpifs** utility. The output from **dumpifs** might look something like this:

```
   Offset     Size  Name
        0      100  Startup-header flags1=0x1 flags2=0 paddr_bias=0x80000000
      100     a008  startup.*
     a108       5c  Image-header mountpoint=/
     a164      264  Image-directory
     ----     ----  Root-dent
     ----       12  usr/lib/ldqnx.so.2 -> /proc/boot/libc.so
     ----        9  dev/console -> /dev/ser1
     a3c8       80  proc/boot/.script
     b000    4a000  proc/boot/procnto
    55000    59000  proc/boot/libc.so.3
     ----        9  proc/boot/libc.so -> libc.so.3
    ae000     7340  proc/boot/devc-ser8250
    b6000     4050  proc/boot/esh
    bb000     4a80  proc/boot/ls
    c0000    14fe0  proc/boot/data1
    d5000     22a0  proc/boot/data2
Checksums: image=0x94b0d37b startup=0xa3aeaf2
```

The more **-v** ("verbose") options you specify to **dumpifs**, the more data you'll see.

For more information on **dumpifs**, see its entry in the *Utilities Reference*.

Building a flash filesystem image

If your application requires a writable filesystem and you have flash memory devices in your embedded system, then you can use a Neutrino flash filesystem driver to provide a POSIX-compatible filesystem. The flash filesystem drivers are described in the Filesystems chapter of the *System Architecture* guide. The chapter on customizing the flash filesystem in this book describes how you can build a flash filesystem driver for your embedded system.

You have two options when creating a flash filesystem:

- Create a flash filesystem image file on the host system and then write the image into the flash on the target.
- Run the flash filesystem driver for your target system, and then copy files into the flash filesystem on the target.

In this section we describe how to create a flash filesystem image file using the **mkefs** (for *mak*e *e*mbedded *f*ilesystem) utility and a buildfile. How to transfer the flash filesystem image onto your target system is described in the "Embedding an image" section. For details on how to use the flash filesystem drivers, see the *Utilities Reference*.

Using mkefs

The **mkefs** utility takes a buildfile and produces a flash filesystem image file. The buildfile is a list of attributes and files to include in the filesystem.

`mkefs` buildfile

The syntax of the buildfile is similar to that for **`mkifs`**, but **`mkefs`** supports a different set of attributes, including the following:

`block_size=`*bsize* Specifies the block size of the flash device being used; defaults to 64 KB. We'll talk about interleave considerations for flash devices below.

`max_size=`*msize* Specifies the maximum size of the flash device; is used to check for overflows. The default is 4 Gbytes.

`spare_blocks=`*sblocks*
Specifies the number of spare blocks to set aside for the flash filesystem; see "Spare blocks," below.

`min_size=`*tsize* Specifies the minimum size of the filesystem. If the resultant image is smaller than *tsize*, the image is padded out to *tsize* bytes. The default is unspecified, meaning that the image won't be padded.

Refer to the *Utilities Reference* for a complete description of the buildfile syntax and attributes supported by **`mkefs`**.

Here's a very simple example of a buildfile:

```
[block_size=128k spare_blocks=1 filter=deflate]
/home/ejm/products/sp1/callp/imagedir
```

In this example, the attributes specify that the flash devices have a block size of 128 KB, that there should be one spare block, and that all the files should be processed using the **`deflate`** utility, which compresses the files. A single directory is given. Just as with **`mkifs`**, when we specify a directory, all files and subdirectories beneath it are included in the resulting image. Most of the other filename tricks shown above for **`mkifs`** also apply to **`mkefs`**.

Block size

The value you should specify for the **`block_size`** attribute depends on the physical block size of the flash device given in the manufacturer's data sheet and on how the flash device is configured in your hardware (specifically the interleave).

Here are some examples:

If you have:	Set *block_size* to:
An 8-bit flash interface and are using an 8-bit device with a 64 KB block size	64 KB
A 16-bit flash interface and are using two interleaved 8-bit flash devices with a 64 KB block size	128 KB
A 16-bit flash interface and are using a 16-bit flash device with a 64 KB block size	64 KB
A 32-bit flash interface and are using four interleaved 8-bit flash devices with a 64 KB block size	256 KB

Notice that you don't have to specify any details (other than the block size) about the actual flash devices used in your system.

Spare blocks

The **spare_blocks** attribute indicates how many blocks should be left as spare. A value of 0 implies a "read/write" (or "write-once") flash filesystem, whereas a value greater than 0 implies a "read/write/reclaim" filesystem.

The default is 1, but the number of spare blocks you'll need depends on the amount of writing you'll do. You should specify an odd number of spare blocks, usually 1 or 3.

The filesystem doesn't use a spare block until it's time to perform a reclaim operation. A nonspare block is then selected for "reclamation", and the data contained in that block is coalesced into one contiguous region in the spare block. The nonspare block is then erased and becomes the new spare block. The former spare block takes the place of the reclaimed block.

If you don't set aside at least one spare block (i.e. the **spare_blocks** attribute is 0), then the flash filesystem driver won't be able to reclaim space — it won't have any place to put the new copy of the data. The filesystem will eventually fill up since there's no way to reclaim space.

Compressing files

The file compression mechanism provided with our flash filesystem is a convenient way to cut flash memory costs for customers. The flash filesystem uses popular deflate/inflate algorithms for fast and efficient compression/decompression.

You can use the **deflate** utility to compress files in the flash filesystem, either from a shell or as the **filter** attribute to **mkefs**. The **deflate** algorithm provides excellent lossless compression of data and executable files.

The flash filesystem drivers use the **inflator** utility to transparently decompress files that have been compressed with **deflate**, which means that you can access compressed files in the flash filesystem without having to decompress them first.

Compressing files can result in significant space savings. But there's a trade-off: it takes longer to access compressed files. Always consider the slowdown of compressed data access and increased CPU usage when designing a system. We've seen systems with restricted flash budget increase their boot time by large factors when using compression.

You can compress files:

- before or as you're using **mkefs** to create the flash filesystem
- to add files to a running flash filesystem

The first method is the high-runner case. You can use the **deflate** utility as a filter for **mkefs** to compress the files that get built into the flash filesystem. For example, you could use this buildfile to create a 16-megabyte filesystem with compression:

```
[block_size=128K spare_blocks=1 min_size=16m filter=deflate]
/bin/
```

You can also *precompress* the files by using **deflate** directly. If **mkefs** detects a compression signature in a file that it's putting into the filesystem, it knows that the file is precompressed, and so it doesn't compress the file again. In either case, **mkefs** puts the data on the flash filesystem and sets a simple bit in the metadata that tells the flash filesystem that the file needs to be decompressed.

The second method is to use **deflate** to compress files and store them directly in the flash filesystem. For example, here's how to use **deflate** at the command line to compress the **ls** file from the image filesystem into a flash filesystem:

```
$ deflate /proc/boot/ls -o /fs0p0/ls
```

Abstraction layer

The flash filesystem never compresses any files. It detects compressed files on the media and uses **inflator** to decompress them *as they're accessed*. An abstraction layer in **inflator** achieves efficiency and preserves POSIX compliance. Special compressed data headers on top of the flash files provide fast seek times.

This layering is quite straightforward. Specific I/O functions include handling the three basic access calls for compressed files:

- *read()*
- *lseek()*
- *lstat()*

Two sizes

This is where compression gets tricky. A compressed file has *two* sizes:

Virtual size — This is, for the end user, the real size of the *decompressed* data, such as *stat()* would report.

Media size — The size that the file actually occupies on the media.

For instance, running the disk usage utility **du** would be practically meaningless under a flash directory with data that is decompressed on the fly. It wouldn't reflect flash media usage at all.

As a convenience, **inflator** supports a naming convention that lets you access the compressed file: simply add **.~~~** (a period and three tildes) to the file name. If you use this extension, the file isn't decompressed, so read operations yield raw compressed data instead of the decompressed data. For example, to get the virtual size of a compressed file, type:

```
ls -l my_file
```

but to get the media size, type:

```
ls -l my_file.~~~
```

Compression rules

If you read a file with the **.~~~** extension, the file isn't decompressed for you, as it would be normally. Now this is where we start talking about rules. All this reading and getting the size of files is fairly simple; things get ugly when it's time to *write* those files.

- You can't write all over the place! Although the flash filesystem supports random writes in uncompressed files, the same isn't true for compressed files.
- Compressed files are read-only; you can replace a compressed file, but you can't modify it in place.
- The flash filesystem never transparently compresses any data.
- If compressed data needs to be put on the flash during the life of a product, this data has to be precompressed.

The exception

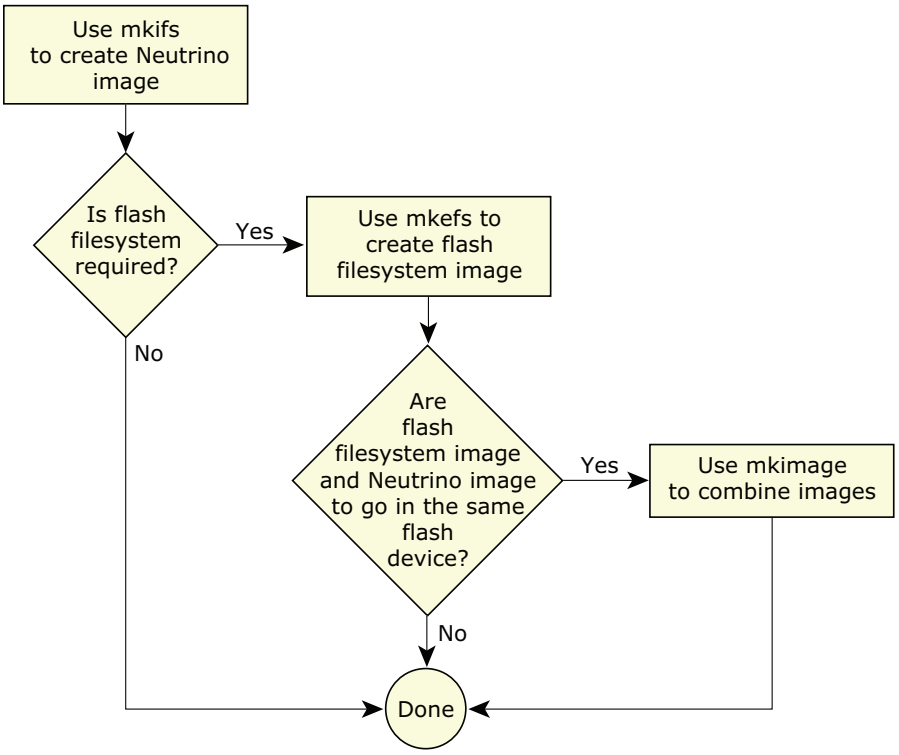
So those are the rules, and here is the exception: truncation. If a compressed file is opened with O_TRUNC from the regular virtual namespace, the file status will become just as if it were created from this namespace. This gives you full POSIX capabilities and no compression with accompanying restrictions.

By the way, the *ftruncate()* functionality isn't provided with compressed files, but is supported with regular files.

Embedding an image

After you've created your bootable OS image on the host system, you'll want to transfer it to the target system so that you can boot Neutrino on the target. The various ways of booting the OS on a target system are described in the chapter on customizing IPL programs in this guide.

If you're booting the OS from flash, then you'll want to write the image into the flash devices on the target. The same applies if you have a flash filesystem image — you'll want to write the image into flash on the target.



Flash configuration options for your Neutrino-based embedded systems.

Depending on your requirements and the configuration of your target system, you may want to embed:

- the IPL
- the boot image
- the boot image and other image filesystem
- the boot image and flash filesystem
- some other combination of the above.

Also, you may wish to write the boot image and the flash filesystem on the same flash device or different devices. If you want to write the boot image and the flash

filesystem on the same device, then you can use the **mkimage** utility to combine the image files into a single image file.

During the initial development stages, you'll probably need to write the image into flash using a programmer or a download utility. Later on if you have a flash filesystem running on your target, you can then write the image file into a raw flash partition.

If your programmer requires the image file to be in some format other than binary, then you can use the **mkrec** utility to convert the image file format.

Combining image files using mkimage

The **mkimage** utility combines multiple input image files into a single output image file. It recognizes which of the image files contains the boot image and will place this image at the start. Note that instead of using **mkimage**, some developers rely on a flash programmer to burn the separate images with appropriate alignment.

For example:

```
mkimage nto.ifs fs.ifs > flash.ifs
```

will take the **nto.ifs** and **fs.ifs** image files and output them to the **flash.ifs** file.

If you want more control over how the image files are combined, you can use other utilities, such as:

- **cat**
- **dd**
- **mkrec**
- **objcopy**

Combining image files using the IDE

You'll use the System Builder to generate OS images for your target board's RAM or flash. You can create:

- an OS image
- a Flash image
- a combined image.

For more information about this process, please see the documentation that comes with the QNX Momentics IDE.

Converting images using mkrec

The **mkrec** utility takes a binary image file and converts it to either Motorola S records or Intel hex records, suitable for a flash or EPROM programmer.

For example:

```
mkrec -s 256k flash.ifs > flash.srec
```

will convert the image file **flash.ifs** to an S-record format file called **flash.srec**. The **-s 256k** option specifies that the EPROM device is 256 KB in size.

If you have multiple image files that you wish to download, then you can first use **mkimage** to combine the image files into a single file before downloading. Or, your flash/EPROM programmer may allow you to download multiple image files at different offsets.

Transferring an image to flash

There are many ways to transfer your image into your flash:

- Use an EPROM burner that supports your socketed flash.
- Use a flash burner that supports onboard flash via a special bus, such as JTAG.
- Use a low-level monitor or a BIOS page with a flash burn command.
- Use the flash filesystem raw mountpoints.

The details on how to transfer the image with anything other than the last method is beyond the scope of this document. Using the raw mountpoint is a convenient way that comes bundled with your flash filesystem library. You can actually read and write raw partitions just like regular files, except that when the raw mountpoint is involved, remember to:

- go down one level in the abstraction ladder
- perform the erase commands yourself.

For the sake of this discussion, we can use the **devf-ram** driver. This driver simulates flash using regular memory. To start it, log in as **root** and type:

```
# devf-ram &
```

You can use the **flashctl** command to erase a partition. You don't need to be **root** to do this. For instance:

```
$ flashctl -p /dev/fs0 -e
```

CAUTION: Be careful when you use this command. Make sure you aren't erasing something important on your flash — like your BIOS!

On normal flash, the **flashctl** command on a raw partition should take a while (about one second for each erase block). This command erases the **/dev/fs0** raw flash array. Try the **hd** command on this newly erased flash array; everything should be **0xFF**:

```
$ hd /dev/fs0
0000000: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
*
```

For more information on **flashctl**, see the *Utilities Reference*.

Let's make a dummy IPL for the purpose of this example:

```
$ echo Hello, World! > ipl
$ mkrec -s 128k -f full ipl > ipl_image
Reset jmps to 0x1FFE0 (jmp 0xFFED)
ROM offset is 0x1FFE0
```

Of course, this IPL won't work for real — it's just for trying out the flash filesystem. In any event, an IPL wouldn't be very useful in RAM. Let's make a dummy flash filesystem for the purpose of this example (the **^D** means Ctrl-D):

```
$ mkefs -v - flash_image
[block_size=128k spare_blocks=1 min_size=384k]
/bin/ls
/bin/cat
^D
writing directory entry ->
writing file entry      -> ls **
writing file entry      -> cat *
Filesystem size = 384K
block size = 128K
1 spare block(s)
```

This flash filesystem actually works (unlike the IPL). Now, the flash partition images can be transferred to the flash using any file-transfer utility (such as **cp** or **ftp**). We have an IPL image created with **mkrec** (and properly padded to an erase block boundary) and a flash image created with **mkefs**, so we can use **cat** to combine and transfer both images to the flash:

```
$ cat ipl_image flash_image > /dev/fs0
```

If you use the **hd** utility on the raw mountpoint again, you'll see that your flash that had initially all bits set to ones (**0xFF**) now contains your partition images. To use the flash filesystem partition, you need to slay the driver and start it again so it can recognize the partitions and mount them. For instance, with **devf-ram**:

```
$ slay devf-ram
$ devf-ram &
```

From this point, you have a **/fs0p1** mountpoint that's in fact a directory and contains the files you specified with **mkefs** to create your flash image. There's no **/fs0p0**, because the boot image isn't recognized by the flash filesystem. It's still accessible as a raw mountpoint via **/dev/fs0p0**. You can do the same operations on **/dev/fs0p0** that you could do with **/dev/fs0**. Even **/dev/fs0p1** is accessible, but be careful not to write to this partition while applications are using the flash filesystem at **/fs0p1**. Try:

```
$ /fs0p1/ls /fs0p1
```

You've just executed **ls** from your flash filesystem and you've listed its contents. To conclude, let's say that what we did in this example is a good starting point for when you customize the flash filesystem to your own platforms. These baby steps should be the first steps to using a full-blown filesystem on your target.

System configuration

In this section, we'll look at some of the ways you can configure Neutrino systems. Please refer to the Sample Buildfiles appendix in this guide for more detailed examples.

What you want to do will, of course, depend on the type of system you're building. Our purpose in this section is to offer some general guidelines and to help clarify which executables should be used in which circumstances, as well as which shared libraries are required for their respective executables.

The general procedure to set up a system is as follows:

1. Establish an output device.
2. Run drivers.
3. Run applications.

Establishing an output device

One of the very first things to do in a buildfile is to start a driver that you then redirect standard input, output, and error to. This allows all subsequent drivers and applications to output their startup messages and any diagnostics messages they may emit to a known place where you can examine the output.

Generally, you'd start either the console driver or a serial port driver. The console driver is used when you're developing on a fairly complete "desktop" type of environment; the serial driver is suitable for most "embedded" environments.

But you may not even have any such devices in your deeply embedded system, in which case you would omit this step. Or you may have other types of devices that you can use as your output device, in which case you may require a specialized driver (that you supply). If you don't specify a driver, output will go to the debug output driver provided by the startup code.

A simple desktop example

This example starts the standard console driver in edited mode (the **-e** option, which is the default). To set up the output device, you would include the driver in your startup script (the **[+script]** file). For example:

```
devc-con -e &
reopen /dev/con1
```

The following starts the **8250** serial port driver in edited mode (the **-e** option), with an initial baud rate of 115200 baud (the **-b** option):

```
devc-ser8250 -e -b115200 &
reopen /dev/ser1
```

In both cases, the **reopen** command causes standard input, output, and error to be redirected to the specified pathname (either **/dev/con1** or **/dev/ser1** in the above

examples). This redirection holds until otherwise specified with another **reopen** command.

The **reopen** used above is a **mkifs** *internal command*, not the shell builtin command of the same name.

Running drivers/filesystems

The next thing you'll want to run are the drivers and/or filesystems that will give you access to the hardware. Note that the console or serial port that we installed in the previous section is actually an example of such a driver, but it was a special case in that it should generally be the first one.

We support several types of drivers/filesystems, including:

- disk drivers (**devb-***)
- flash filesystems (**devf-***)
- network drivers (**devn-***, **devnp-***)
- input drivers (**devi-***)
- USB drivers (**devu-***)
- filesystems (**fs-***)

Which one you install first is generally driven by where your executables reside. One of the goals for the image is to keep it small. This means that you generally don't put all the executables and shared libraries you plan to load directly into the image — instead, you place those files into some other medium (whether a flash filesystem, rotating disk, or a network filesystem). In that case, you should start the appropriate driver to get access to your executables. Once you have access to your executables on some medium, you would *then* start other drivers from that medium.

The alternative, which is often found in deeply embedded systems, is to put all the executables and shared libraries *directly into the image*. You might want to do this if there's no secondary storage medium or if you wanted to have everything available immediately, without the need to start a driver.

Let's examine the steps required to start the disk, flash, and network drivers. All these drivers share a common feature: they rely on one process that loads one or more **.so** files, with the particular **.so** files selected either via the command line of the process or via automatic configuration detection.

Since the various drivers we're discussing here use **.so** files (not just their own driver-specific ones, but also standard ones like the C library), these **.so** files must be present *before* the driver starts. Obviously, this means that the **.so** file *cannot* be on the same medium as the one you're trying to start the driver for! We recommend that you put these **.so** files into the image filesystem.

Disk drivers

The first thing you need to determine is which hardware you have controlling the disk interface. We support a number of interfaces, including various flavors of SCSI controllers and the EIDE controller. For details on the supported interface controllers, see the various **devb-*** entries in the *Utilities Reference*.

The only action required in your buildfile is to start the driver (e.g. **devb-aha7**). The driver will then dynamically load the appropriate modules (in this order):

1. **libcam.so** — Common Access Method library
2. **cam-*.so** — Common Access Method module(s)
3. **io-blk.so** — block I/O module
4. **fs-*.so** — filesystem personality module(s)

The CAM **.so** files are documented under **cam-*** in the *Utilities Reference*. Currently, we support CD-ROMs (**cam-cdrom.so**), hard disks (**cam-disk.so**), and optical disks (**cam-optical.so**).

The **io-blk.so** module is responsible for dealing with a disk on a block-by-block basis. It includes caching support.

The **fs-*** modules are responsible for providing the high-level knowledge about how a particular filesystem is structured. We currently support the following:

Filesystem	Module
MS-DOS	**fs-dos.so**
Linux	**fs-ext2.so**
Macintosh HFS and HFS Plus	**fs-mac.so**
Windows NT	**fs-nt.so**
QNX 4	**fs-qnx4.so**
Power-Safe	**fs-qnx6.so**
ISO-9660 CD-ROM, Universal Disk Format (UDF)	**fs-udf.so**

Flash filesystems

To run a flash filesystem, you need to select the appropriate flash driver for your target system. For details on the supported flash drivers, see the various **devf-*** entries in the *Utilities Reference*.

The **devf-generic** flash driver that can be thought of as a universal driver whose capabilities make it accessible to most flash devices.

The flash filesystem drivers don't rely on any flash-specific **.so** files, so the only module required is the standard C library (**libc.so**).

Since the flash filesystem drivers are written for specific target systems, you can usually start them without command-line options; they'll find the flash for the specific system they were written for.

Network drivers

Network services are started from the **io-pkt*** command, which is responsible for loading in the required **.so** files.

For dynamic control of network drivers, you can simply use **mount** and **umount** to start and stop drivers at the command line. For example:

```
mount -T io-pkt devn-ne2000.so
```

For more information, see **mount** in the *Utilities Reference*.

Two levels of **.so** files are started, based on the command-line options given to **io-pkt***:

- **-d** specifies driver **.so** files
- **-p** specifies protocol **.so** files.

The **-d** option lets you choose the hardware driver that knows how to talk to a particular card. For example, choosing **-d ne2000** will cause **io-pkt*** to load **devn-ne2000.so** to access an NE-2000-compatible network card. You may specify additional command-line options after the **-d**, such as the interrupt vector to be used by the card.

The **-p** option lets you choose the protocol driver that deals with a particular protocol. As with the **-d** option, you would specify command-line options after the **-p** for the driver, such as the IP address for a particular interface.

For more information about network services, see the **devn-***, and **io-pkt** entries in the *Utilities Reference*.

Network filesystems

We support two types of network filesystems:

- NFS (**fs-nfs2**, **fs-nfs3**), which allows file access over a network to a UNIX or other system running an NFS server.
- CIFS (**fs-cifs**), which allows file access over a network to a Windows 98 or NT system or to a UNIX system running an SMB server.

The CIFS protocol makes no attempt to conform to POSIX.

Although NFS is primarily a UNIX-based filesystem, you may find some versions of NFS available for Windows.

Running applications

There's nothing special required to run your applications. Generally, they'll be placed in the script file *after* all the other drivers have started. If you require a particular driver to be present and "ready," you would typically use the **waitfor** command in the script.

Here's an example. An application called **peelmaster** needs to wait for a driver (let's call it **driver-spud**) to be ready before it should start. The following sequence is typical:

```
driver-spud &
waitfor /dev/spud
peelmaster
```

This causes the driver (**driver-spud**) to be run in the background (specified by the ampersand character). The expectation is that when the driver is ready, it will register the pathname **/dev/spud**. The **waitfor** command tries to *stat()* the pathname **/dev/spud** periodically, blocking execution of the script until the pathname appears or a predetermined timeout has occurred. Once the pathname appears in the pathname space, we assume that the driver is ready to accept requests. At that point, the **waitfor** will unblock, and the next program in the list (in our case, **peelmaster**) will execute.

Without the **waitfor** command, the **peelmaster** program would run immediately after the driver was started, which could cause **peelmaster** to miss the **/dev/spud** pathname and fail.

Debugging an embedded system

When you're developing embedded systems under some operating systems, you often need to use a *hardware debugger*, a physical device that connects to target hardware via a JTAG (Joint Test Action Group) interface. This is necessary for development of drivers, and possibly user applications, because they're linked into the same memory space as the kernel. If a driver or application crashes, the kernel and system may crash

as a result. This makes using software debuggers difficult, because they depend on a running system.

Debugging target systems with Neutrino is different because its architecture is significantly different from other embeddable realtime operating systems:

- All Neutrino applications (including drivers) run in their own memory-protected virtual address space. This has the advantage that the software is more reliable and fault tolerant. However, conventional hardware debuggers rely on decoding physical memory addresses, making them incompatible with debugging user applications based in a virtual memory environment.
- Neutrino lets you develop multithreaded applications, which hardware debuggers generally don't support.

Under Neutrino, you typically use:

- a hardware debugger for the IPL and startup
- a software debugger for the rest of the software

In other words, you rarely have to use a JTAG hardware debugger, especially if you're using one of our board support packages.

`pdebug` software debugging agent

We provide a software debugging agent called **`pdebug`** that makes it easier for you to debug system drivers and user applications. The **`pdebug`** agent runs on the target system and communicates with the host debugger over a serial or Ethernet connection.

For more information, see "The process-level debug agent" in the Compiling and Debugging chapter of the *Programmer's Guide*.

Hardware debuggers and Neutrino

The major constraint of using **`pdebug`** is that the kernel must already be running on the target. In other words, you can't use **`pdebug`** until the IPL and startup have successfully started the kernel.

However, the IPL and startup program run with the CPU in physical mode, so you can use conventional hardware debuggers to debug them. This is the primary function of the JTAG debugger throughout the Neutrino software development phase. You use the hardware debugger to debug the BSP (IPL and startup), and **`pdebug`** to debug drivers and applications once the kernel is running. You can also use a hardware debugger to examine registers and view memory while the kernel and applications are running, if you know the physical addresses.

If hardware debuggers, such as SH or AMC have builtin Neutrino awareness, you can use a JTAG to debug applications. These debuggers can interpret kernel information as well as perform the necessary translation between virtual and physical memory addresses to view application data.

Producing debug symbol information for IPL and startup

You can use hardware debuggers to debug Neutrino IPL and startup programs without any extra information. However, in this case, you're limited to assembly-level debugging, and assembler symbols such as subroutine names aren't visible. To perform full source-level debugging, you need to provide the hardware debugger with the symbol information and C source code.

This section describes the steps necessary to generate the symbol and debug information required by a hardware debugger for source-level debugging. The steps described are based on the PPC (PowerPC) Board Support Package available for Neutrino 6.3.0 for both IPL and startup of the Motorola Sandpoint MPC750 hardware reference platform.

The examples below are described for a Neutrino 6.3 self-hosted environment, and assume that you're logged in on the development host with **root** privileges.

Generating IPL debug symbols

To generate symbol information for the IPL, you must recompile both the IPL library and the Sandpoint IPL with debug information. The general procedure is as follows:

1. Modify the IPL source.
2. Build the IPL library and Sandpoint IPL.
3. Burn the IPL into the flash memory of the Sandpoint board using a flash burner or JTAG.
4. Modify the **sandpoint.lnk** file to output ELF format.
5. Recompile the IPL library and Sandpoint IPL source with debug options.
6. Load the Sandpoint IPL ELF file containing debug information into the hardware debugger.

Be sure to synchronize the source code, the IPL burned into flash, and the IPL debug symbols.

To build the IPL library with debug information:

```
# cd bsp_working_dir/src/hardware/ipl/lib/ppc/a.be
# make clean
# make CCOPTS=-g
# cp libipl.a bsp_working_dir/sandpoint/install/ppcbe/lib
# make install
```

The above steps recompile the PowerPC IPL library (**libipl.a**) with DWARF debug information and copy this library to the Sandpoint install directory. The Sandpoint BSP is configured to look for this library first in its install directory. The **make install** is optional, and copies **libipl.a** to **/ppcbe/usr/lib**.

The Sandpoint BSP has been set up to work with SREC format files. However, to generate debug and symbol information to be loaded into the hardware debugger, you must generate ELF-format files.

Modify the **sandpoint.lnk** file to output ELF format:

```
# cd bsp_working_dir/sandpoint/src/hardware/ipl/boards/sandpoint
```

Edit the file **sandpoint.lnk**, changing the first lines from:

```
TARGET(elf32-powerpc)
OUTPUT_FORMAT(srec)
ENTRY(entry_vec)
```

to:

```
TARGET(elf32-powerpc)
OUTPUT_FORMAT(elf32-powerpc)
ENTRY(entry_vec)
```

You can now rebuild the Sandpoint IPL to produce symbol and debug information in ELF format. To build the Sandpoint IPL with debug information:

```
# cd bsp_working_dir/sandpoint/src/hardware/ipl/boards/sandpoint/ppc/be
# make clean
# make CCOPTS=-g
```

The **ipl-sandpoint** file is now in ELF format with debug symbols from both the IPL library and Sandpoint IPL.

To rebuild the BSP, you need to change the **sandpoint.lnk** file back to outputting SREC format. It's also important to keep the IPL that's burned into the Sandpoint flash memory in sync with the generated debug information; if you modify the IPL source, you need to rebuild the BSP, burn the new IPL into flash, and rebuild the IPL symbol and debug information.

You can use the **objdump** utility to view the ELF information. For example, to view the symbol information contained in the **ipl-sandpoint** file:

```
# objdump -t ipl-sandpoint | less
```

You can now import the **ipl-sandpoint** file into a hardware debugger to provide the symbol information required for debugging. In addition, the hardware debugger needs the source code listings found in the following directories:

- *bsp_working_dir*/sandpoint/src/hardware/ipl/boards/sandpoint
- *bsp_working_dir*/src/hardware/ipl/lib
- *bsp_working_dir*/src/hardware/ipl/lib/ppc

Generating startup debug symbols

To generate symbol information for startup, you must recompile both the startup library and the Sandpoint startup with debug information. The general procedure is as follows:

1. Modify the startup source.
2. Build the startup library and Sandpoint startup with debug information.
3. Rebuild the image and symbol file.
4. Load the symbol file into the hardware debugger program.
5. Transfer the image to the Sandpoint target (burn into flash, transfer over a serial connection).

To build the startup library with debug information:

```
# cd bsp_working_dir/src/hardware/startup/lib/ppc/a.be
# make clean
# make CCOPTS=-g
# cp libstartup.a bsp_working_dir/sandpoint/install/ppcbe/lib
# make install
```

The above steps recompile the PowerPC startup library (**libstartup.a**) with DWARF debug information and copy this library to the Sandpoint install directory. The Sandpoint BSP is configured to look for this library first in its install directory. The **make install** is optional, and copies **libstartup.a** to **/ppcbe/usr/lib**.

To build the Sandpoint startup with debugging information:

```
# cd bsp_working_dir/sandpoint/src/hardware/startup/boards/sandpoint/ppc/be
# make clean
# make CCOPTS=-g
# make install
```

The above steps generate the file startup-sandpoint with symbol and debug information. Again, you can use the **-gstabs+** debug option instead of **-g**. The **make install** is necessary, and copies **startup-sandpoint** into the Sandpoint install directory, *bsp_working_dir***/sandpoint/install/ppcbe/boot/sys**.

You can't load the startup-sandpoint ELF file into the hardware debugger to obtain the debug symbols, because the **mkifs** utility adds an offset to the addresses defined in the symbols according to the offset specified in the build file.

Modify the build file to include the **+keeplinked** attribute for startup:

```
# cd bsp_working_dir/sandpoint/images
```

Modify the startup line of your build file to look like:

```
[image=0x10000]
[virtual=ppcbe,binary +compress] .bootstrap = {
        [+keeplinked] startup-sandpoint -vvv -D8250
    PATH=/proc/boot procnto-600 -vv
}
```

The **+keeplinked** option makes **mkifs** generate a symbol file that represents the debug information positioned within the image filesystem by the specified offset.

To rebuild the image to generate the symbol file:

```
# cd bsp_working_dir/sandpoint/images
# make clean
```

Then, if you're using one of the provided **.build** files:

```
# make all
```

otherwise:

```
# mkifs -v -r ../install myfile.build image
```

These commands create the symbol file, **startup-sandpoint.sym**. You can use the **objdump** utility to view the ELF information.

To view the symbol information contained in the **startup-sandpoint.sym** file:

```
# objdump -t startup-sandpoint.sym | less
```

You can now import the **startup-sandpoint.sym** file into a hardware debugger to provide the symbol information required for debugging startup. In addition, the hardware debugger needs the source code listings found in the following directories:

- *bsp_working_dir*/src/hardware/startup/lib
- *bsp_working_dir*/src/hardware/startup/lib/public/ppc
- *bsp_working_dir*/src/hardware/startup/lib/public/sys
- *bsp_working_dir*/src/hardware/startup/lib/ppc
- *bsp_working_dir*/sandpoint/src/hardware/startup/boards/sandpoint

Chapter 4

Writing an IPL Program

In this chapter. . .

Initial program loader (IPL)

In this section, we'll examine the IPL program in detail, including how to customize it for your particular hardware, if you need to.

Responsibilities of the IPL

The initial task of the IPL is to minimally configure the hardware to create an environment that allows the startup program (e.g. **`startup-bios`**, **`startup-ixdp425`**, etc.), and consequently the Neutrino microkernel, to run. This includes at least the following:

1. Start execution from the reset vector.
2. Configure the memory controller. This may include configuring the chip selects and/or PCI controller.
3. Configure clocks.
4. Set up a stack to allow the IPL library to perform OS verification and setup (download, scan, set up, and jump to the OS image).

The IPL's initialization part is written entirely in assembly language (because it executes from ROM with no memory controller). After initializing the hardware, the IPL then calls the *main()* function to initiate the C-language environment.

Once the C environment is set up, the IPL can perform different tasks, depending on whether the OS is booting from a linearly mapped device or a bank-switched device:

Linearly mapped	The entire image is in the processor's linear address space.
Bank-switched	The image isn't entirely addressable by the processor (e.g. bank-switched ROM, disk device, network, etc.).

Note that we use the term "ROM" generically to mean any nonvolatile memory device used to store the image (Flash, RAM, ROM, EPROM, flash, battery-backed SRAM, etc.).

Linearly mapped images

For linearly mapped images, we have the following sources:

- ROM

Bank-switched images

For bank-switched images, we have the following sources:

- PC-Card (PCMCIA) (some implementations)
- ROM, RAM, bank-switched
- Network device

- Serial or parallel port
- Disk device
- Other.

Processors & configurations

In conjunction with the above, we have the following processors and configurations:

- 386 and higher processors, which power up in 16-bit real mode.
- PowerPC family of processors, (some are physical and some are virtual processors), which power up in 32-bit physical or virtual mode.
- ARM family of processors (StrongARM, XScale), which power up in 32-bit physical mode.
- MIPS architecture processors, which power up with virtual addressing enabled, but mapped one-to-one.
- SH-4 family of processors, which power up with virtual addressing enabled, but mapped one-to-one.

Booting from a bank-switched device

Let's assume we're booting from a bank-switched or paged device (e.g. paged flash, disk device, network, etc.), and that the image is uncompressed. The IPL needs to handle these main tasks:

1. The IPL must first use a C function to talk to the device in question. We'll use a serial download for this discussion. For serial downloads, the IPL uses *image_download_8250()*, a function that specifically knows how to configure and control the 8250 class of serial controllers.

   Once the controller is set up, the function's task is to copy the image via the serial controller to a location in RAM.

2. We now have an OS image in RAM. The IPL then uses the *image_scan()* function, which takes a start address and end address as its parameters. It returns the address at which it found the image:

   `unsigned long image_scan (unsigned long` *start*`,` **`unsigned long`** *end*`)`

   The *image_scan()* function:

   - Scans for a valid OS signature over the range provided. Note that this can be multiple OS images.
   - Copies the startup header from the image to a **`struct`** *startup_header* variable.
   - Authenticates the startup signature (STARTUP_HDR_SIGNATURE).
   - Performs a checksum on the startup.
   - Performs a checksum on the OS image filesystem.

- Saves the address and version number of the OS in case it's set up to scan for multiple OS images.

3 Once the OS image has been found and validated, the IPL's next function to call is *image_setup()*, which takes the address of the image as its parameter and always returns 0:

```
int image_setup (unsigned long address)
```

The *image_setup()* function:

- Copies the startup header from the image to a **`struct startup_header`** variable. Although this was performed in *image_scan()* (and **`startup_header`** is a global), it's necessary here because *image_scan()* can scan for multiple images, which will overwrite this structure.
- Calculates the address to which startup is to be copied, based on the *ram_paddr* and *paddr_bias* structure members (from the startup header).
- Fills in the *imagefs_paddr* structure member, based on where the image is stored. The startup program relies on this member, because it's the one responsible for copying the OS image filesystem to its final location in RAM. The startup program doesn't necessarily know where the image is stored.
- Copies the final startup structure to the *ram_paddr* address, and then copies the startup program itself.

At this phase, the startup program has been copied to RAM (and it must *always* execute from RAM), and the startup header has been patched with the address of the OS image.

Since the startup program is responsible for copying the image filesystem to its final destination in RAM, the IPL must copy the image to a location that's *linearly accessible* by the startup program, which has no knowledge of paged devices (serial, disk, parallel, network, etc.).

Note also that if the image is compressed, then the IPL can copy the compressed image to a location that won't interfere with startup's decompression of the image to its final destination in RAM. When the image lives in flash (or ROM or whatever linear storage device), this isn't an issue. But when the image is stored on a paged device, more care must be taken in placing the image in a RAM location that won't interfere with startup's decompression of the image. Here are the rules:

Uncompressed	If the image is uncompressed, then the IPL can copy the image from the paged device directly to its destined location. Startup will compare the addresses and realize that the image doesn't need to be copied.
Compressed	If the image is compressed, then startup must copy and decompress the image *using a different location* than the final RAM location.

4 The last phase is to jump to the startup entry point. This is accomplished by calling *image_start()*:

```
int image_start (unsigned long address)
```

The *image_start()* function should never return; it returns -1 if it fails.

The function jumps to the *startup_vaddr* address as defined in the startup header.

Booting from a linear device

For a system that boots from a linearly mapped device (e.g. linear flash, ROM, etc.), the IPL's tasks are the same as in the paged-device scenario above, but with one notable exception: the IPL doesn't need to concern itself with copying a full OS image from the device to RAM.

"Warm" vs "cold" start

Your IPL code may be quite simple or fairly elaborate, depending on how your embedded system is configured. We'll use the terms *warm start* and *cold start* to describe the different types of IPL:

Warm-start IPL	If there's a BIOS or ROM monitor already installed at the reset vector, then your IPL code is simply an extension to the BIOS or ROM monitor.
Cold-start IPL	The system doesn't have (or doesn't use) a BIOS or ROM monitor program. The IPL must be located at the reset vector.

Warm-start IPL

In this case, the IPL doesn't get control immediately after the reset, but instead gets control from the BIOS or ROM monitor.

The x86 PC BIOS allows extensions, as do various ROM monitors. During the power-up memory scan, the BIOS or ROM monitor attempts to detect extensions in the address space. To be recognized as an extension, the extension ROM must have a well-defined *extension signature* (e.g. for a PC BIOS, this is the sequence **`0x55`** and then **`0xAA`** as the first two bytes of the extension ROM). The extension ROM must be prepared to receive control at the *extension entry offset* (e.g. for a PC BIOS, this is an offset of **`0x0003`** into the extension ROM).

Note that this method is used by the various PC BOOTP ROMs available. The ROM presents itself as an extension, and then, when control is transferred to it, gets an image from the network and loads it into RAM.

Cold-start IPL

One of the benefits of Neutrino, especially in a cost-reduced embedded system, is that you don't *require* a BIOS or ROM monitor program. This discussion is primarily for developers who must write their own IPL program or who (for whatever reason) don't wish to use the default IPL supplied by their BIOS/monitor.

Let's take a look at what the IPL does in this case.

When power is first applied to the processor (or whenever the processor is reset), some of its registers are set to a known state, and it begins executing from a known memory location (i.e. the *reset vector*).

Your IPL software must be located at the reset vector and must be able to:

1. Set up the processor.
2. Locate the OS image.
3. Copy the startup program into RAM.
4. Transfer control to the startup program.

For example, on an x86 system, the reset vector is located at address **`0xFFFFFFF0`**. The device that contains the IPL must be installed within that address range. In a typical x86 PC BIOS, the reset vector code contains a **`JMP`** instruction that then branches to the code that performs diagnostics, setup, and IPL functionality.

Loading the image

Regardless of the processor being used, once the IPL code is started, it has to load the image in a manner that meets the requirements of the Neutrino microkernel as described above. The IPL code may also have to support a backup way of loading the image (e.g. an **`.altboot`** in the case of a hard/floppy boot). This may also have to be an automatic fallback in the case of a corrupted image.

Note, however, that the amount of work your IPL code has to do really depends on the location of the image; there may be only a small amount of work for the IPL or there may be a lot.

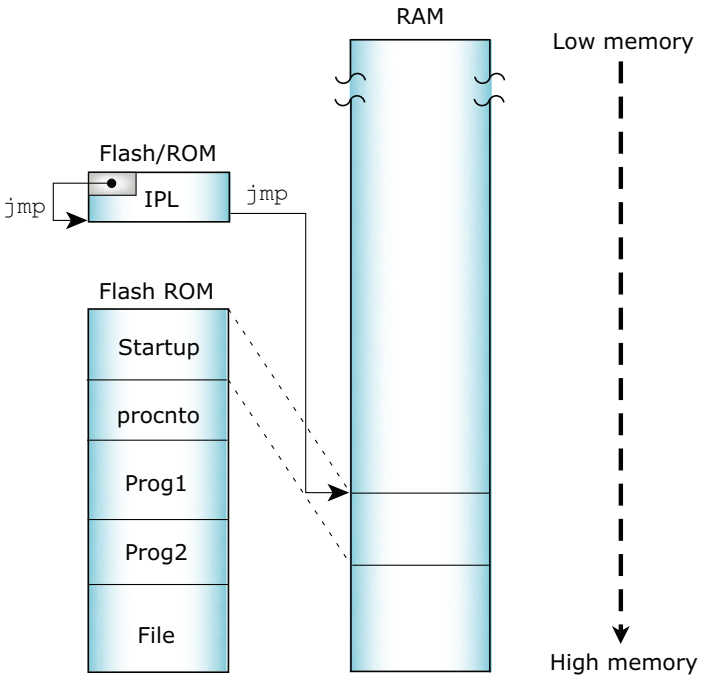
Let's look again at the two classifications of image sources.

If the source is a linearly mapped device

This is the simplest scenario. In this case, the entire image is stored in some form of directly addressable storage — either a ROM device or a form of PC-Card device that maps its entire address space into the processor's address space. All that's required is to copy the startup code into RAM. This is ideal for small or deeply embedded systems.

Note that on x86 architectures, the device *isn't* required to be addressable within the first megabyte of memory. The startup program also needn't be in the first megabyte of RAM.

Note also that for PC-Card devices, some form of setup may be required before the entire PC-Card device's address space will appear in the address space of the processor. It's up to your IPL code to perform this setup operation. (We provide library routines for several standard PC-Card interface chips.)



Linearly mapped device.

If the source is a bank-switched device

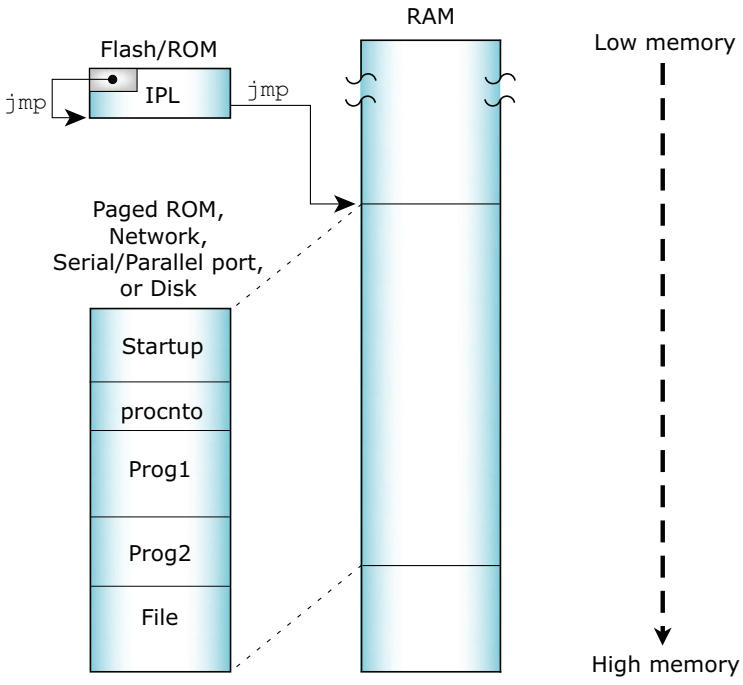
In this scenario, the image is stored in a device that isn't directly mapped into linear memory. An additional factor needs to be considered here — how will your IPL code get at the image stored in the device?

Many types of hardware devices conform to this model:

- ROM
- Network boot
- Serial or parallel port
- Traditional disk

Let's look at the common characteristics. In such systems, the IPL code knows how to fetch data from some piece of hardware. The process is as follows:

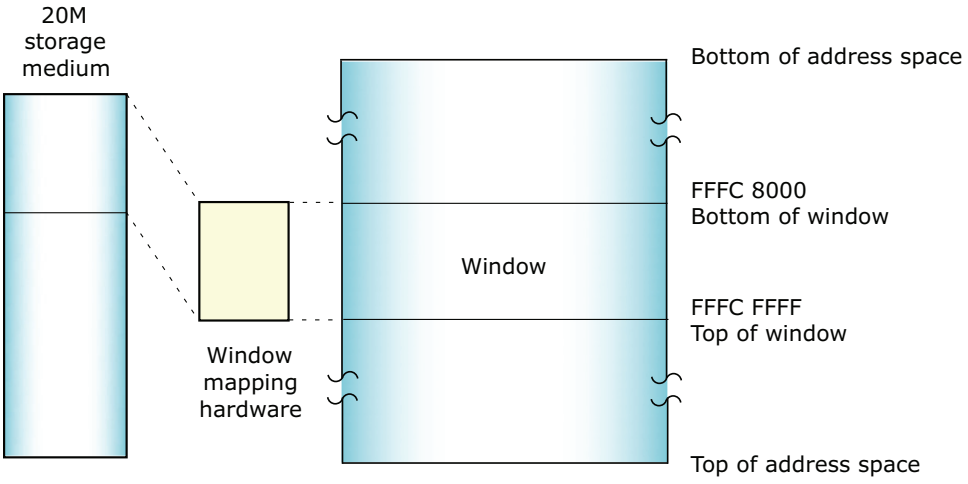
1. The IPL receives control.
2. The IPL loads the image from the hardware into RAM.
3. The IPL then transfers control to the newly loaded image.



Bank-switched devices.

ROM devices

In this scenario, a solid-state storage device (ROM, EPROM, flash, etc.) contains the image, but the processor can see only a small portion of the contents of the device. How is this implemented? The hardware has a small window (say 32 KB) into the address space of the processor; additional hardware registers control which portion of the device is manifested into that window.



Large storage medium, bank-switched into a window.

In order to load the image, your IPL code must know how to control the hardware that maps the window. Your IPL code then needs to copy the image out of the window into RAM and transfer control.

If possible, avoid the use of any mapping hardware (whether custom-designed or "industry-standard") — it only serves to complicate the hardware and software designs. We strongly recommend linearly mapped devices. (See the appendix on System Design Considerations for more information.)

Network boot

Depending on your embedded system's requirements or on your development process, you can load the image via an Ethernet network. On some embedded boards, the ROM monitor contains the BOOTP code. On a PC with an ISA or PCI network card, some form of boot ROM is placed into the address space of the processor, where we assume the PC BIOS will transfer control to it. The BOOTP code knows how to talk to the networking hardware and how to get the image from a remote system.

Using a BOOTP server

To boot a Neutrino system using BOOTP, you'll need a BOOTP ROM for your OS client and a BOOTP server (e.g. **`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 (Neutrino, UNIX, Windows 95/98/NT.)

Serial port

A serial port on the target can be useful during development for downloading an image or as a failsafe mechanism (e.g. if a checksum fails, you can simply reload the image via the serial port).

A serial loader can be built into the IPL code so that the code can fetch the image from an external hardware port. This generally has a minimal impact on the cost of an embedded system; in most cases, the serial port hardware can be left off for final assembly. Evaluation boards supplied by hardware chip vendors often have serial ports. We supply source code for an embedded serial loader for the 8250 chip.

The IPL process in this case is almost identical to the one discussed above for the Network boot, except that the serial port is used to fetch the image.

Traditional disk

In a traditional PC-style embedded system with a BIOS, this is the simplest boot possible. The BIOS performs all the work for you — it fetches the image from disk, transfers it to RAM, and starts it.

On the other hand, if you don't have a BIOS but you wish to implement this kind of a boot, then this method involves the most complicated processing discussed so far. This is because you'll need a driver that knows how to access the disk (whether it's a traditional rotating-medium hard disk or a solid-state disk). Your IPL code then needs to look into the partition table of the device and figure out where the contents of the image reside. Once that determination has been made, the IPL then needs to either map the image portions into a window and transfer bytes to RAM (in the case of a solid-state disk) or fetch the data bytes from the disk hardware.

None of the above?

It's entirely conceivable that none of the above adequately describes your particular embedded system. In that case, the IPL code you'll write must still perform the same basic steps as described above — handle the reset vector, fetch the image from some medium, and transfer control to the startup routine.

Transferring control to the startup program

Once the image has either been loaded into RAM or is available for execution in ROM, we must transfer control to the *startup code* (copied from the image to RAM).

For detailed information about the different types of startup programs, see the chapter on Customizing Image Startup Programs.

Once the startup code is off and running, the work of the IPL process is done.

Customizing IPLs

This section describes in detail the steps necessary to write the IPL for an embedded system that boots from ROM or Flash.

Systems that boot from disk or over the network typically come with a BIOS or ROM monitor, which already contains a large part of the IPL within it. If your embedded system fits this category, you can probably skip directly to the chapter on Customizing Image Startup Programs.

Your IPL loader gets control at reset time and performs the following main functions:

1. Initialize hardware (via assembly-language code).
2. Download the image into RAM (e.g. via serial using *image_download_8250()*).
3. Locate the OS image (via *image_scan()*).
4. Copy the startup program (via *image_setup()*).
5. Jump to the loaded image (via *image_start()*).

Initialize hardware

Basic hardware initialization is done at this time. This includes gaining access to the system RAM, which may not be addressable after reset. The amount of initialization done here will depend on what was done by any code before this loader gained control. On some systems, the power-on-reset will point directly to this code, which will have to do everything. On other systems, this loader may be called by an even more primitive loader, which may have already performed some of these tasks.

Note that it's not necessary to initialize standard peripheral hardware such as an IDE interface or the baud rate of serial ports. This will be done by the OS drivers when they're started later. Technically, you need to initialize only enough hardware to allow control to be transferred to the startup program in the image.

The startup program is written in C and is provided in full source-code format. The startup code is structured in a readily customizable manner, providing a simple environment for performing further initializations, such as setting up the *system page* in-memory data structure.

Loading the image into RAM

The IPL code must locate the boot image (made with the **`mkifs`** utility) and copy part or all of it into memory.

The loader uses information in the header to copy the *header* and *startup* into RAM. The loader would be responsible for copying the entire image into RAM if the image weren't located in linearly addressable memory.

Structure of the boot header

The boot header structure **`struct startup_header`** is defined in the include file **`<sys/startup.h>`**. It is 256 bytes in size and contains the following members, which are examined by the IPL and/or startup code:

`unsigned long` *signature*
`unsigned short` *version*
`unsigned char` *flags1*
`unsigned char` *flags2*
`unsigned short` *header_size*
`unsigned short` *machine*
`unsigned long` *startup_vaddr*
`unsigned long` *paddr_bias*
`unsigned long` *image_paddr*
`unsigned long` *ram_paddr*
`unsigned long` *ram_size*
`unsigned long` *startup_size*
`unsigned long` *stored_size*
`unsigned long` *imagefs_paddr*
`unsigned long` *imagefs_size*
`unsigned short` *preboot_size*
`unsigned short` *zero0*
`unsigned long` *zero* **`[3]`**
`unsigned long` *info* **`[48]`**

A valid image (for bootable images) is detected by performing a checksum (via the function call *checksum()*) over the entire image, as follows:

```
checksum (image_paddr, startup_size);
checksum (image_paddr + startup_size, stored_size - startup_size);
```

signature

This is the first 32 bits in the header and always contains **`0x00FF7EEB`** in native byte order. It's used to identify the header. On a machine that can be either big-endian or little-endian (a *bi-endian* machine, e.g. MIPS), there's typically a hardware strap that gets set on the board to specify the endianness.

version

The version of **`mkifs`** that made the image.

flags1 and *flags2*

The following flags are defined for *flags1* (*flags2* is currently not used):

STARTUP_HDR_FLAGS1_VIRTUAL

If this flag is set, the operating system is to run with the Memory Management Unit (MMU) enabled.

For this release of Neutrino, you should always specify a virtual system (by specifying the **`virtual=`** attribute in your buildfile, which then sets the STARTUP_HDR_FLAGS1_VIRTUAL flag).

STARTUP_HDR_FLAGS1_BIGENDIAN

The processor is big-endian. Processors should always examine this flag to check that the ENDIAN is right for them.

STARTUP_HDR_FLAGS1_COMPRESS_NONE

The image isn't compressed.

STARTUP_HDR_FLAGS1_COMPRESS_ZLIB

The image is compressed using **`libz`** (**`gzip`**).

STARTUP_HDR_FLAGS1_COMPRESS_LZO

The image is compressed with **`liblzo`**.

STARTUP_HDR_FLAGS1_COMPRESS_UCL

The image is compressed with **`libucl`**. This is the format chosen when using the **`[+compress]`** attribute in the **`mkifs`** build script.

Currently, the **`startup-*`** programs are built to understand only the UCL compression method. By twiddling the SUPPORT_CMP_* macro definitions in **`startup/lib/uncompress.c`**, you can change to one of the other supported compression methods.

The STARTUP_HDR_FLAGS1_COMPRESS_* constants aren't really flags because they may set more than one bit; they're used as an enumeration of the types of compression.

Note that both flag *flags1* and *flags2* are single-byte; this ensures that they're endian-neutral.

header_size

The size of the startup header (**`sizeof (struct startup_header)`**).

machine

Machine type, from **`<sys/elf.h>`**.

startup_vaddr

Virtual address to transfer to after IPL is done.

paddr_bias

Value to add to physical address to get a value to put into a pointer and indirect through.

image_paddr

The physical address of the image. This can be in ROM or RAM, depending on the type of image; for more information, see "Relationship of **`struct startup_header`** fields," later in this chapter.

ram_paddr

The physical address in RAM to copy the image to. You should copy *startup_size* bytes worth of data.

ram_size

The number of bytes the image will occupy when it's loaded into RAM. This value is used by the startup code in the image and isn't currently needed by the IPL code. This size may be greater than *stored_size* if the image was compressed. It may also be smaller than *stored_size* if the image is XIP.

startup_size

This is the size of the startup code. Copy this number of bytes from the start of the image into RAM. Note that the startup code is never compressed, so this size is true in all cases.

stored_size

This is the size of the image including the header. The *stored_size* member is also used in the copy/decompress routines for non-XIP images.

imagefs_paddr

Set by the IPL to the physical address of the image filesystem. Used by the startup.

imagefs_size

Size of uncompressed image filesystem.

preboot_size

Contains the number of bytes from the beginning of the loaded image to the startup header. Note that this value will usually be zero, indicating that nothing precedes the startup portion. On an x86 with a BIOS, it will be nonzero, because there's a small

piece of code that gets data from the BIOS in real mode and then switches into protected mode and performs the startup.

zero and *zero0*

Zero filler; reserved for future expansion.

info

An array of **`startup_info*`** structures. This is the communications area between the IPL and the startup code. When the IPL code detects various system features (amount of memory installed, current time, information about the bus used on the system, etc.), it stores that information into the *info* array so that the startup code can fetch it later. This saves the startup code from performing the same detection logic again.

Note that the *info* is declared as an array of **`long`**s — this is purely to allocate the storage space. In reality, the *info* storage area contains a set of structures, each beginning with this header:

```
struct startup_info_hdr {
    unsigned short  type;
    unsigned short  size;
};
```

The *type* member is selected from the following list:

STARTUP_INFO_SKIP

Ignore this field. If the corresponding *size* member is 0, it means that this is the end of the *info* list.

STARTUP_INFO_MEM

A **`startup_info_mem`** or **`startup_info_mem_extended`** structure is present.

STARTUP_INFO_DISK

A **`startup_info_disk`** structure is present.

STARTUP_INFO_TIME

A **`startup_info_time`** structure is present.

STARTUP_INFO_BOX

A **`startup_info_box`** structure is present.

Note that the **`struct startup_info_hdr`** header (containing the *type* and *size* members) is encapsulated within each of the above mentioned **`struct startup_info*`** structures as the first element.

Let's look at the individual structures.

struct startup_info_skip

Contains only the header as the member *hdr*.

struct startup_info_mem and startup_info_mem_extended

These structures contain an address and size pair defining a chunk of memory that should be added to **procnto**'s free memory pool.

The **startup_info_mem** structure is defined as follows:

```
struct startup_info_mem {
    struct startup_info_hdr     hdr;
    unsigned long               addr;
    unsigned long               size;
};
```

The *addr* and *size* fields are 32 bits long, so memory is limited to 4 GB. For larger memory blocks, the **startup_info_mem_extended** structure is used:

```
struct startup_info_mem_extended {
    struct startup_info_mem     mem;
    unsigned long               addr_hi;
    unsigned long               size_hi;
};
```

For the extended structure, determine the address and size from the *addr_hi* and *size_hi* members and the encapsulated **startup_info_mem** structure as follows:

```
((paddr64_t) addr_hi << 32) | mem.addr
((paddr64_t) size_hi << 32) | mem.size
```

More than one **startup_info_mem** or **startup_info_mem_extended** structure may be present to accommodate systems that have free memory located in various blocks throughout the address space.

Both these structures are indentified by a *type* member of STARTUP_INFO_MEM in the **startup_info_hdr** structure; use the *size* field in the header to tell them apart.

struct startup_info_disk

Contains the following:

```
struct startup_info_disk {
    struct startup_info_hdr     hdr;
    unsigned char               drive;
    unsigned char               zero;
    unsigned short              heads;
    unsigned short              cylinders;
    unsigned short              sectors;
    unsigned long               blocks;
};
```

Contains information about any hard disks detected (on a PC with a BIOS). The members are as follows:

drive Drive number.

zero	Reserved; must be zero.
heads	Number of heads present.
cylinders	Number of cylinders present.
sectors	Number of sectors present.
blocks	Total blocksize of device. Computed by the formula *heads* × *cylinders* × *sectors*. Note that this assumes 512 bytes per block.

`struct startup_info_time`

Contains the following:

```
struct startup_info_time {
    struct startup_info_hdr     hdr;
    unsigned long               time;
};
```

The *time* member contains the current time as the number of seconds since 1970 01 01 00:00:00 GMT.

`struct startup_info_box`

Contains the following:

```
struct startup_info_box {
    struct startup_info_hdr     hdr;
    unsigned char               boxtype;
    unsigned char               bustype;
    unsigned char               spare [2];
};
```

Contains the *boxtype* and *bustype* information. For valid values, please see the chapter on Customizing Image Startup Programs.

The *spare* fields are reserved and must be zero.

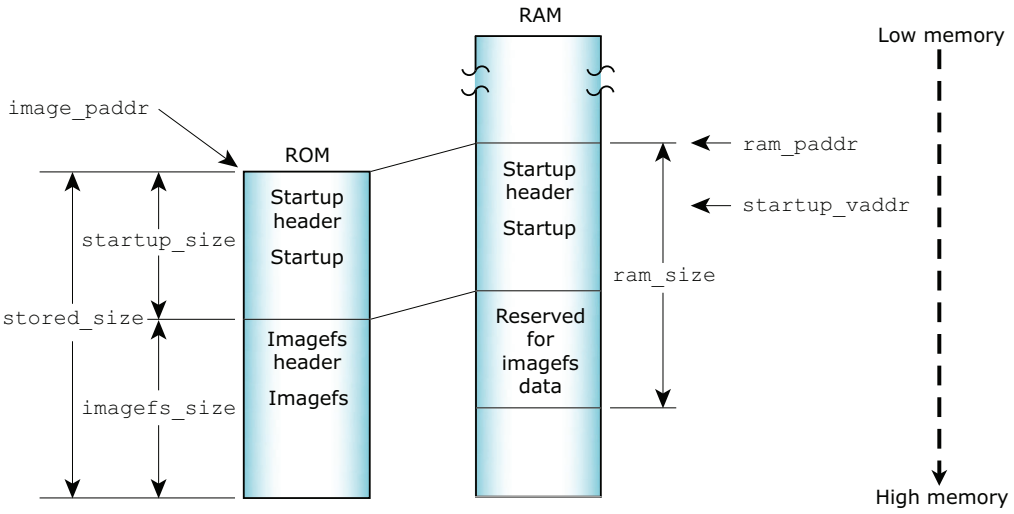
Relationship of `struct startup_header` fields

The following explains some of the fields used by the IPL and startup for various types of boot. These fields are stuffed by **`mkifs`**.

Note that we've indicated which steps are performed by the IPL and which are done by the startup.

Linear ROM execute-in-place boot image

The following illustration shows an XIP image:



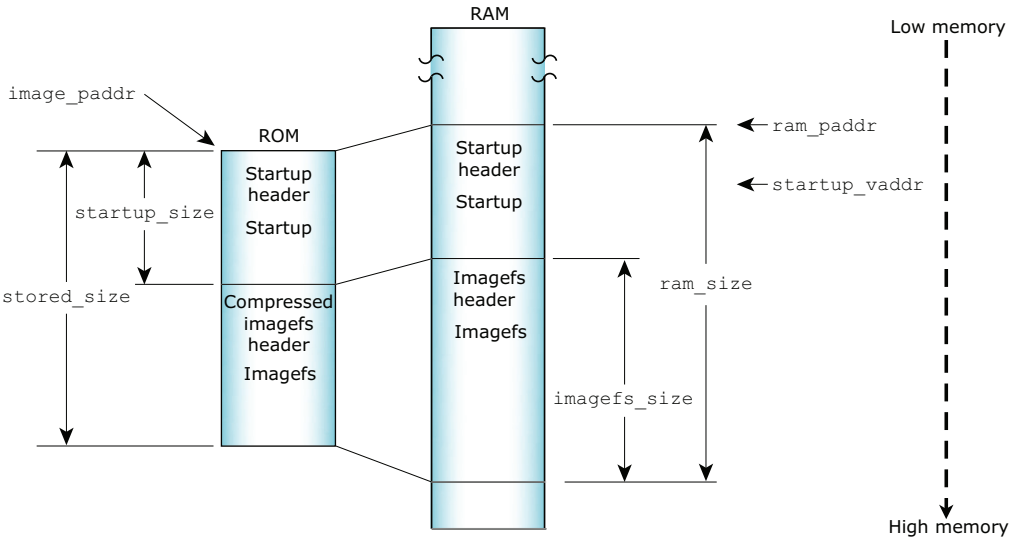
In the following pseudo-code examples, *image_paddr* represents the source location of the image in linear ROM, and *ram_paddr* represents the image's destination in RAM.

Here are the steps required in the IPL:

```
checksum (image_paddr, startup_size)
checksum (image_paddr + startup_size, stored_size - startup_size)
copy (image_paddr, ram_paddr, startup_size)
jump (startup_vaddr)
```

Linear ROM compressed boot image

Here's the same scenario, but with a compressed image:



Here are the steps required in the IPL:

```
checksum (image_paddr, startup_size)
```

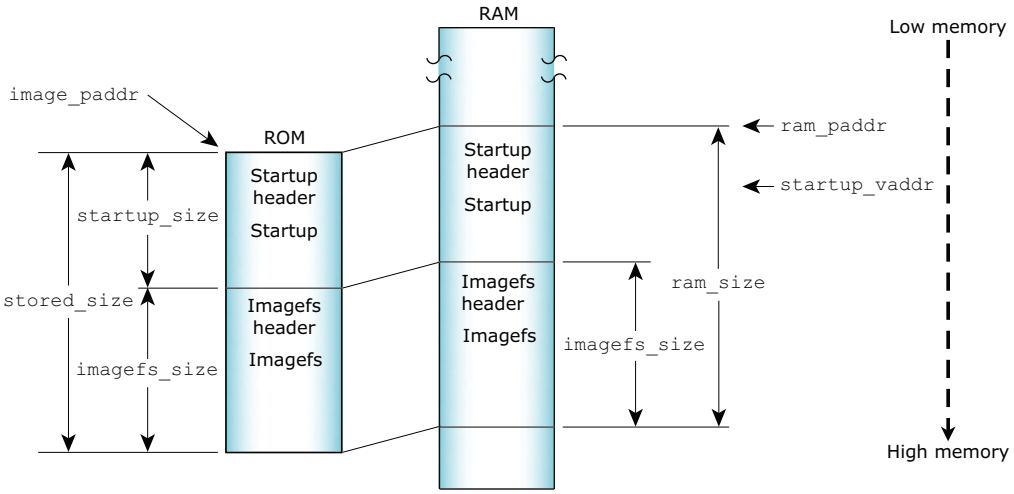
```
checksum (image_paddr + startup_size, stored_size - startup_size)
copy (image_paddr, ram_paddr, startup_size)
jump (startup_vaddr)
```

And here's the step required in the startup:

```
uncompress (ram_paddr + startup_size, image_paddr + startup_size,
            stored_size - startup_size)
```

ROM non-XIP image

In this scenario, the image doesn't execute in place:



Here are the steps required in the IPL:

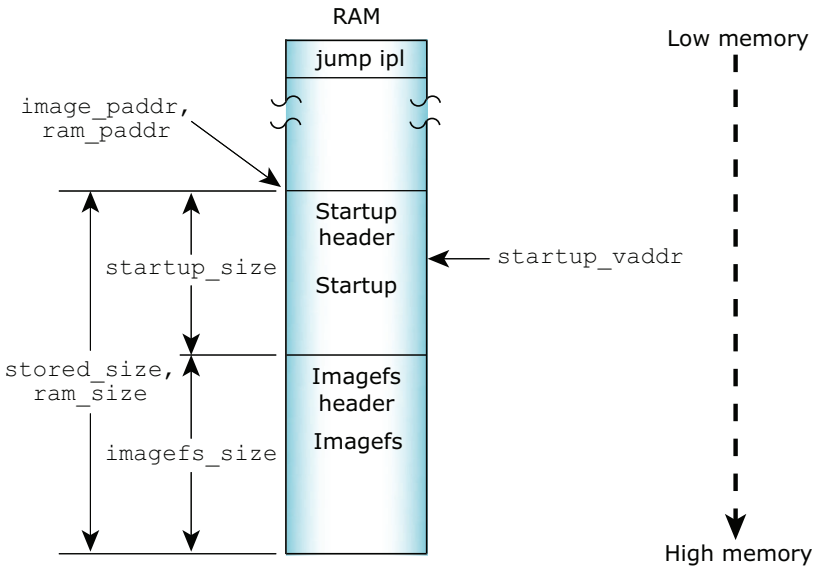
```
checksum (image_paddr, startup_size)
checksum (image_paddr + startup_size, stored_size - startup_size)
copy (image_paddr, ram_paddr, startup_size)
jump (startup_vaddr)
```

And here's the step required in the startup:

```
copy (ram_paddr + startup_size, image_paddr + startup_size,
      stored_size - startup_size)
```

Disk/network image (x86 BIOS)

In this case our full IPL isn't involved. An existing BIOS IPL loads the image into memory and transfers control to our IPL. Since the existing IPL doesn't know where in startup to jump, it always jumps to the start of the image. On the front of the image we build a tiny IPL that jumps to *startup_vaddr*:

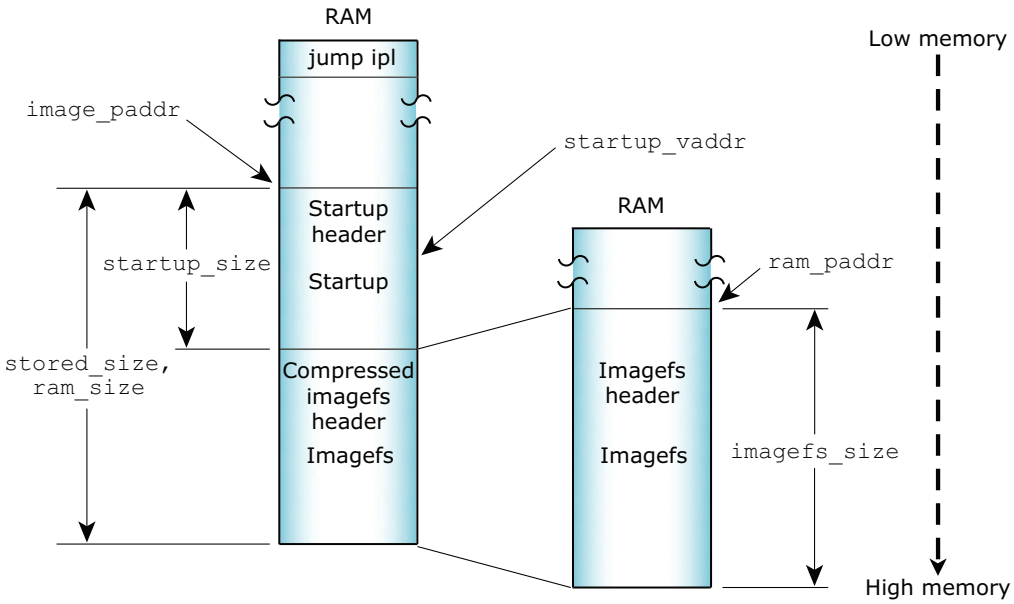


Here's the step required in the IPL:

```
jump (startup_vaddr)
```

Disk/network compressed image

This is identical to the previous case, except that we need to decompress the image in the startup:



Here's the step required in the startup:

```
uncompress (ram_paddr + startup_size, image_paddr + startup_size,
            stored_size - startup_size)
```

The case of a bank-switched ROM is much like a disk/network boot except you get to write the code that copies the image into RAM using the following steps in the IPL:

```
bankcopy (image_paddr, ram_paddr, startup_size)
checksum (image_paddr, startup_size)
checksum (image_paddr + startup_size, stored_size - startup_size)
jump (startup_vaddr)
```

Your next step is to go to the disk/network or disk/network compressed scenario above.

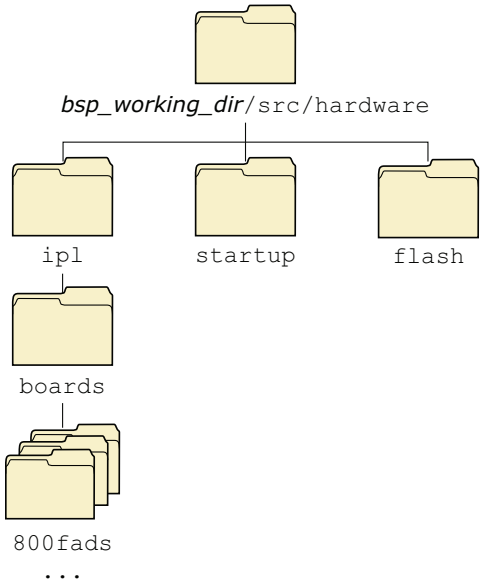
You'll need to map the physical addresses and sizes into bank-switching as needed. Have fun and next time **don't bank-switch your rom!** Make it linear in the address space.

IPL structure

In this section, we'll examine the structure of the IPL source tree directory, and also the structure of a typical IPL source file.

IPL source directory structure

The Neutrino source tree structure looks like this:



IPL directory structure.

The *bsp_working_dir*`/src/hardware/ipl/boards` directory is where the IPL source code is stored for a particular board (e.g. *bsp_working_dir*`/src/hardware/ipl/boards/800fads` contains the source code for the Motorola MPC8xxFADS PowerPC motherboard.)

IPL code structure

The IPL code is structured in two stages. The first stage is written in assembly language; it sets up just enough of an environment for the second stage, written in C, to run. Generally, the minimum work done here is to set up the DRAM controllers,

initialize the various registers, and set up the chip selects so that you can address your hardware.

Generally, the IPL assembly-language source name begins with "**init**" (e.g. **init8xx.s** for the MPC8xxFADS board); the C file is always called **main.c**.

Once your assembly-language routine has set up the minimum amount required to transfer control to the C language portion, the *main()* program calls the following functions in order:

image_download_8250()

This function is responsible for getting the image from wherever it may be located. If the image is located in linear memory, this function isn't required (the image is already "downloaded").

If you're downloading the image from a custom piece of hardware, you should call your function *image_download_hw()*, where the *hw* part is replaced with a descriptive name for the hardware, e.g. *image_download_x25()*.

image_scan() This function is given a start and an end address to search for a boot image. If successful, it returns a pointer to the start of the image. It's possible to search within an address range that contains more than one image. If there are multiple images, and one of them has a bad checksum, then the next image is used. If there are multiple images with good checksums, the startup header is examined, and the one with the higher version number is used. Note that the scan will occur *only* between the specified addresses.

image_setup() This function does the work of copying the necessary part of the image into RAM.

image_start() This function will jump to the start of the image loaded into RAM, which will turn control over to the startup program.

An example

Take the **main.c** from the FADS8xx system:

```
#include "ipl.h"

unsigned int image;

int
main (void)
{
/*
 * Image is located at 0x2840000
 * Therefore, we don't require an image_download_8250 function
 */
    image = image_scan (0x2840000, 0x2841000);

/*
```

```
 * Copy startup to ram; it will do any necessary work on the image
 */
    image_setup (image);

/*
 * Set up link register and jump to startup entry point
 */
    image_start (image);

    return (0);
}
```

In this case, we have a linearly addressable flash memory device that contains the image — that's why we don't need the *image_download_8250()* function.

The next function called is *image_scan()*, which is given a very narrow range of addresses to scan for the image. We give it such a small range because we *know* where the image is on this system — there's very little point searching for it elsewhere.

Then we call *image_setup()* with the address that we got from the *image_scan()*. This copies the startup code to RAM.

Finally, we call *image_start()* to transfer control to the startup program. We don't expect this function to return — the reason we have the **`return (0);`** statement is to keep the C compiler happy (otherwise it would complain about "Missing return value from function main").

Creating a new IPL

To create a new IPL, it's best to start with one we've provided that's similar to the type of CPU and board you have in your design.

The basic steps are:

1. Create a new directory under *bsp_working_dir*`/src/hardware/ipl/boards` with your board name.
2. Copy all files and subdirectories from a similar board into the new directory.
3. Modify the files as appropriate.

The IPL library

The IPL library contains a set of routines for building a custom IPL. Here are the available library functions:

Function	Description
enable_cache	Enable the on-chip cache (x86 only).
image_download_8250()	Download an image from the specified serial port.

continued...

Function	Description
image_scan()	Scan memory for a valid system image.
image_scan_ext()	BIOS extension version of *image_scan()*.
image_setup()	Prepare an image for execution.
image_setup_ext()	BIOS extension version of *image_setup()*.
image_start()	Transfer control to the image.
image_start_ext()	BIOS extension version of *image_start()*.
int15_copy()	Copy data from high (above 1 MB) memory to a buffer or to low (below 1 MB) memory (x86 only).
print_byte()	Print a byte to video (x86 only).
print_char()	Print a character to video (x86 only).
print_long()	Print a long to video (x86 only).
print_sl()	Print a string, followed by a long to video (x86 only).
print_string()	Print a string to video (x86 only).
print_var()	Print a variable to video (x86 only).
print_word()	Print a word to video (x86 only).
protected_mode	Switch the processor to protected mode (x86 only).
uart_hex8	Output an 8-bit hex number to the UART (x86 only).
uart_hex16	Output a 16-bit hex number to the UART (x86 only).
uart_hex32	Output a 32-bit hex number to the UART (x86 only).
uart_init	Initialize the on-chip UART (x86 only).
uart_put	Output a single character to the UART (x86 only).
uart_string	Output a NULL-terminated string to the UART (x86 only).
uart32_hex8	Output an 8-bit hex number to the UART (for 32-bit protected mode environment; x86 only).
uart32_hex16	Output a 16-bit hex number to the UART (for 32-bit protected mode environment; x86 only).
uart32_hex32	Output a 32-bit hex number to the UART (for 32-bit protected mode environment; x86 only).
uart32_init	Initialize the on-chip UART (for 32-bit protected mode environment; x86 only).

continued...

Function	Description
uart32_put	Output a single character to the UART (for 32-bit protected mode environment; x86 only).
uart32_string	Output a NULL-terminated string to the UART (for 32-bit protected mode environment; x86 only).

enable_cache

```
enable_cache
```

The *enable_cache()* function takes no parameters. The function is meant to be called before the x86 processor is switched to protected mode. Note that the function is for a non-BIOS system.

image_download_8250()

```
unsigned int image_download_8250 (port, span, address)
```

Downloads an image from the specified serial port (*port*) to the specified address (*address*) using a custom protocol. On the host side, this protocol is implemented via the utility **sendnto** (you may need a NULL-modem cable — the protocol uses only TX, RX, and GND). The *span* parameter indicates the offset from one port to the next port on the serial device.

image_scan()

```
unsigned long image_scan (unsigned long start, unsigned long end)
```

The *image_scan()* function scans memory for a valid system image. It looks on 4 KB boundaries for the image identifier bytes and then does a checksum on the image.

The function scans between *start* and *end*. If a valid image is found, *image_scan()* returns the image's address. If no valid image is found, it returns -1.

Note that *image_scan()* will search for *all* images within the given range, and will pick the "best" one as described above (in the "IPL code structure" section).

image_scan_ext()

```
unsigned long image_scan_ext (unsigned long start, unsigned long end)
```

This is a BIOS extension version of the *image_scan()* function. The *image_scan_ext()* function operates in a 16-bit real-mode environment.

image_setup()

```
int image_setup (unsigned long address)
```

The *image_setup()* function prepares an image for execution. It copies the RAM-based startup code from ROM.

The function takes the image's address as its parameter and always returns 0.

image_setup_ext()

```
int image_setup_ext (unsigned long address)
```

This is a BIOS extension version of the *image_setup()* function. The *image_setup_ext()* function operates in a 16-bit real-mode environment and makes use of the *int15_copy()* function to perform its tasks on the OS image.

image_start()

```
int image_start (unsigned long address)
```

The *image_start()* function starts the image by jumping to the *startup_vaddr* address as defined in the startup header.

The function should never return; if it fails, it returns -1.

image_start_ext()

```
int image_start_ext (unsigned long address)
```

This is a BIOS extension version of the *image_start()* function. The *image_start_ext()* function operates in a 16-bit real-mode environment.

int15_copy()

```
unsigned char int15_copy (long from, long to, long len)
```

The *int15_copy()* function is intended for an x86 system with a BIOS running in real mode. The function lets you copy data from high memory (above 1 MB) to a buffer or to low memory (below 1 MB).

The *int15_copy()* function also allows functions such as *image_scan()* and *image_setup()* to perform scanning and setup of images living in high memory.

print_byte()

```
void print_byte (int n)
```

Using int10, this function displays a byte to video (x86 only).

print_char()

```
void print_char (int c)
```

Using int10, this function displays a character to video (x86 only).

print_long()

```
void print_long (unsigned long n)
```

Using int10, this function displays a long to video (x86 only).

print_sl()

```
void print_sl (char *s, unsigned long n)
```

Using int10, this function displays to video a string, followed by a long (x86 only).

print_string()

```
void print_string (char *msg)
```

Using int10, this function displays a string to video (x86 only).

print_var()

```
void print_var (unsigned long n, int l)
```

Using int10, this function displays a variable to video (x86 only).

print_word()

```
void print_word (unsigned short n)
```

Using int10, this function displays a word to video (x86 only).

protected_mode()

This assembly call switches the x86 processor into protected mode. The function is for non-BIOS systems.

Upon return, the DS and ES registers will be set to selectors that can access the entire 4 GB address space. This code is designed to be completely position-independent.

This routine must be called with a pointer to a 16-byte area of memory that's used to store the GDT. The pointer is in **ds:ax**.

The following selectors are defined:

8 Data selector for 0-4 GB.

16 Code selector for 0-4 GB.

uart_hex8

This assembly call outputs an 8-bit hex number to the UART. The function is set up for a 16-bit real-mode environment (x86 only).

On entry:

DX UART base port.

AL Value to output.

uart_hex16

This assembly call outputs a 16-bit hex number to the UART. The function is set up for a 16-bit real-mode environment (x86 only).

On entry:

DX UART base port.

AX Value to output.

uart_hex32

This assembly call outputs a 32-bit hex number to the UART. The function is set up for a 16-bit real-mode environment (x86 only).

On entry:

DX UART base port.

EAX Value to output.

uart_init

This assembly call initializes the on-chip UART to 8 data bits, 1 stop bit, and no parity (8250 compatible). The function is set up for a 16-bit real-mode environment (x86 only).

On entry:

EAX Baud rate.

EBX Input clock in Hz (normally 1843200).

ECX UART internal divisor (normally 16).

DX UART base port.

uart_put

This assembly call outputs a single character to the UART. The function is set up for a 16-bit real-mode environment (x86 only).

On entry:

AL Character to output.

DX UART base port.

uart_string

This assembly call outputs a NULL-terminated string to the UART. The function is set up for a 16-bit real-mode environment (x86 only).

On entry:

DX UART base port address, return address, string.

For example:

```
mov         UART_BASE_PORT, %dx
call        uart_string
.ascii      "string\r\n"
...
```

uart32_hex8

This assembly call outputs an 8-bit hex number to the UART. The function is set up for a 32-bit protected-mode environment (x86 only).

On entry:

DX UART base port.

AL Value to output.

uart32_hex16

This assembly call outputs a 16-bit hex number to the UART. The function is set up for a 32-bit protected-mode environment (x86 only).

On entry:

DX UART base port.

AX Value to output.

uart32_hex32

This assembly call outputs a 32-bit hex number to the UART. The function is set up for a 32-bit protected-mode environment (x86 only).

On entry:

DX UART base port.

EAX Value to output.

uart32_init

This assembly call initializes the on-chip UART to 8 data bits, 1 stop bit, and no parity (8250 compatible). The function is set up for a 32-bit protected-mode environment (x86 only).

On entry:

EAX Baud rate.

EBX Input clock in Hz (normally 1843200).

ECX UART internal divisor (normally 16).

DX UART base port.

uart32_put

This assembly call outputs a single character to the UART. The function is set up for a 32-bit protected-mode environment (x86 only).

On entry:

AL Character to output.

DX UART base port.

uart32_string

This assembly call outputs a NULL-terminated string to the UART. The function is set up for a 32-bit protected-mode environment (x86 only).

On entry:

DX UART base port address, return address, string.

For example:

```
mov         UART_BASE_PORT, %dx
call        uart_string
.ascii      "string\r\n"
...
```

Chapter 5

Customizing Image Startup Programs

In this chapter. . .

Introduction

The first program in a bootable Neutrino image is a *startup program* whose purpose is to:

1. Initialize the hardware.
2. Initialize the system page.
3. Initialize callouts.
4. Load and transfer control to the next program in the image.

You can customize Neutrino for different embedded-system hardware by changing the startup program.

Initialize hardware

You do basic hardware initialization at this time. The amount of initialization done here will depend on what was done in the IPL loader.

Note that you don't need to initialize standard peripheral hardware such as an IDE interface or the baud rate of serial ports. This will be done by the drivers that manage this hardware when they're started.

Initialize system page

Information about the system is collected and placed in an in-memory data structure called the system page. This includes information such as the processor type, bus type, and the location and size of available system RAM.

The kernel as well as applications can access this information as a read-only data structure. The hardware/system-specific code to interrogate the system for this information is confined to the startup program. This code doesn't occupy any system RAM after it has run.

Initialize callouts

Another key function of the startup code is that the system page callouts are *bound in*. These callouts are used by the kernel to perform various hardware- and system-specific functions that must be specified by the systems integrator.

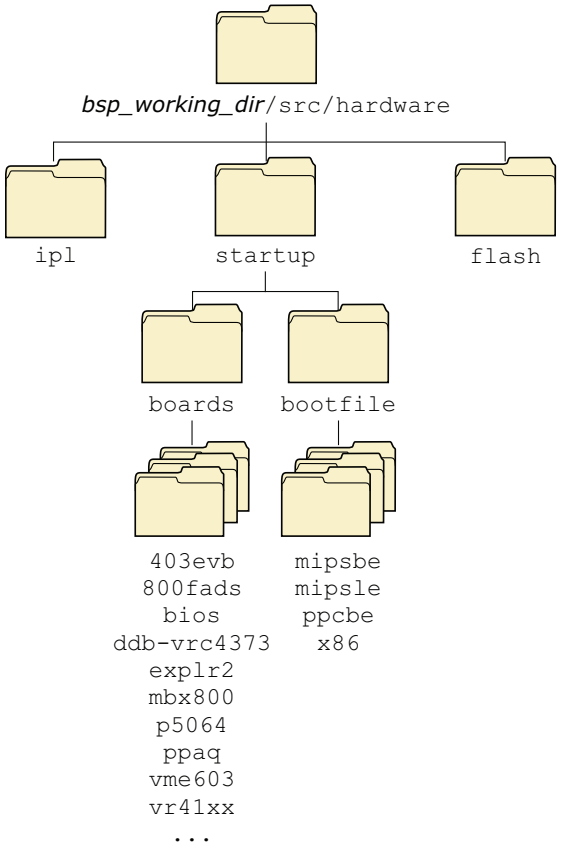
Anatomy of a startup program

Each release of Neutrino ships with a growing number of startup programs for many boards. To find out what boards we currently support, please refer to the following sources:

- the **`boards`** directory under *bsp_working_dir*`/src/hardware/startup`
- QNX docs (BSP docs as well as **`startup-*`** entries in the *Utilities Reference*)

- the Community area of our website, **www.qnx.com**

Each startup program is provided as a ready-to-execute binary. Full source and a Makefile are also available so you can customize and remake each one. The files are kept in this directory structure as illustrated:



Startup directory structure.

Generally speaking, the following directory structure applies in the startup source for the **startup-***boardname* module:

*bsp_working_dir***/src/hardware/startup/boards/***boardname*

Structure of a startup program

Each startup program consists of a *main()* with the following structure (in pseudo code):

```
Global variables

main()
{
    Call add_callout_array()
```

```
    Argument parsing (Call handle_common_option())

    Call init_raminfo()
    Remove ram used by modules in the image

    if (virtual) Call init_mmu() to initialize the MMU

    Call init_intrinfo()
    Call init_qtime()
    Call init_cacheattr()
    Call init_cpuinfo()

    Set hardware machine name

    Call init_system_private()

    Call print_syspage() to print debugging output
}
```

You should examine the commented source for each of the functions within the library to see if you need to replace a library function with one of your own.

Creating a new startup program

To create a new startup program, you should make a new directory under *bsp_working_dir*`/src/hardware/startup/boards` and copy the files from one of the existing startup program directories. For example, to create something close to the Intel PXA250TMDP board, called `my_new_board`, you would:

1. `cd` *bsp_working_dir*`/src/hardware/startup/boards`
2. `mkdir my_new_board`
3. `cp -r pxa250tmdp/* my_new_board`
4. `cd my_new_board`
5. `make clean`

For descriptions of all the startup functions, see "The startup library" section in this chapter.

Structure of the system page

As mentioned earlier (see the section "Initialize system page"), one of the main jobs of the startup program is to initialize the *system page*.

The system page structure `struct syspage_entry` is defined in the include file `<sys/syspage.h>`. The structure contains a number of constants, references to other structures, and a union shared between the various processor platforms supported by Neutrino.

It's important to realize that there are two ways of accessing the data within the system page, depending on whether you're adding data to the system page at startup time or

reading data from the system page later (as would be done by an application program running after the system has been booted). Regardless of which access method you use, the fields are the same.

Here's the system page structure definition, taken from **<sys/syspage.h>**:

```
/*
 * contains at least the following:
 */
struct syspage_entry {
    uint16_t             size;
    uint16_t             total_size;
    uint16_t             type;
    uint16_t             num_cpu;
    syspage_entry_info   system_private;
    syspage_entry_info   asinfo;
    syspage_entry_info   hwinfo;
    syspage_entry_info   cpuinfo;
    syspage_entry_info   cacheattr;
    syspage_entry_info   qtime;
    syspage_entry_info   callout;
    syspage_entry_info   callin;
    syspage_entry_info   typed_strings;
    syspage_entry_info   strings;
    syspage_entry_info   intrinfo;
    syspage_entry_info   smp;
    syspage_entry_info   pminfo;

    union {
        struct x86_syspage_entry     x86;
        struct ppc_syspage_entry     ppc;
        struct mips_syspage_entry    mips;
        struct arm_syspage_entry     arm;
        struct sh_syspage_entry      sh;
    } un;
};
```

Note that some of the fields presented here may be initialized by the code provided in the startup library, while some may need to be initialized by code provided by you. The amount of initialization required really depends on the amount of customization that you need to perform.

Let's look at the various fields.

size

The size of the system page entry. This member is set automatically by the library.

total_size

The size of the system page entry *plus* the referenced substructures; effectively the size of the entire system-page database. This member is set automatically by the library and adjusted later (grown) as required by other library calls.

type

This is used to indicate the CPU family for determining which union member in the *un* element to use. Can be one of:
SYSPAGE_ARM, SYSPAGE_MIPS, SYSPAGE_PPC, SYSPAGE_SH4, or SYSPAGE_X86.

The library sets this member automatically.

num_cpu

The *num_cpu* member indicates the number of CPUs present on the given system. This member is initialized to the default value `1` in the library and adjusted by the library call *init_smp()* if additional processors are detected.

system_private

The *system_private* area contains information that the operating system needs to know when it boots. This is filled in by the startup library's *init_system_private()* function.

Member	Description
user_cpupageptr	User address (R/O) for *cpupage* pointer
user_syspageptr	User address (R/O) for *syspage* pointer
kern_cpupageptr	Kernel address (R/W) for *cpupage* pointer
kern_syspageptr	Kernel address (R/W) for *syspage* pointer
pagesize	Granularity of the OS memory allocator (usually 16 in physical mode or 4096 in virtual mode).

asinfo

The *asinfo* section consists of an array of the following structure. Each entry describes the attributes of one section of address space on the machine.

```
struct asinfo_entry {
    uint64_t            start;
    uint64_t            end;
    uint16_t            owner;
    uint16_t            name;
    uint16_t            attr;
    uint16_t            priority;
    int                 (*alloc_checker)(struct syspage_entry *__sp,
                                         uint64_t   *__base,
                                         uint64_t   *__len,
                                         size_t     __size,
                                         size_t     __align);
    uint32_t            spare;
};
```

Member	Description
start	Gives the first physical address of the range being described.
end	Gives the last physical address of the range being described. Note that this is the actual last byte, *not* one beyond the end.
owner	An offset from the start of the section giving the owner of this entry (its "parent" in the tree). It's set to AS_NULL_OFF if the entry doesn't have an owner (it's at the "root" of the address space tree).
name	An offset from the start of the *strings* section of the system page giving the string name of this entry.
attr	Contains several bits affecting the address range (see below).
priority	Indicates the speed of the memory in the address range. Lower numbers mean slower memory. The macro AS_PRIORITY_DEFAULT is defined to use a default value for this field (currently defined as 100).

The *alloc_checker* isn't currently used. When implemented, it will let you provide finer-grain control over how the system allocates memory (e.g. making sure that ISA memory used for DMA doesn't cross 64 KB boundaries).

The *attr* field

The *attr* field can have the following bits:

`#define AS_ATTR_READABLE 0x0001`

Address range is readable.

`#define AS_ATTR_WRITABLE 0x0002`

Address range is writable.

`#define AS_ATTR_CACHABLE 0x0004`

Address range can be cached (this bit should be off if you're using device memory).

`#define AS_ATTR_KIDS 0x0010`

Indicates that there are other entries that use this one as their owner. Note that the library turns on this bit automatically; you shouldn't specify it when creating the section.

`#define AS_ATTR_CONTINUED 0x0020`

Indicates that there are multiple entries being used to describe one "logical" address range. This bit will be on in all but the last one. Note that the library turns on this bit and uses it internally; you shouldn't specify it when creating the section.

Address space trees

The *asinfo* section contains trees describing address spaces (where RAM, ROM, flash, etc. are located).

The general hierarchy for address spaces is:

/memory/*memclass***/....**

Or:

/io/*memclass***/....**

Or:

/memory/io/*memclass***/....**

The **memory** or **io** indicates whether this is describing something in the memory or I/O address space (the third form is used on a machine without separate in/out instructions and where everything is memory-mapped).

The *memclass* is something like: **ram**, **rom**, **flash**, etc. Below that would be further classifications, allowing the process manager to provide typed memory support.

hwinfo

The *hwinfo* area contains information about the hardware platform (type of bus, devices, IRQs, etc). This is filled in by the startup library's *init_hwinfo()* function.

This is one of the more elaborate sections of the Neutrino system page. The *hwinfo* section doesn't consist of a single structure or an array of the same type. Instead, it consists of a sequence of symbolically "tagged" structures that as a whole describe the hardware installed on the board. The following types and constants are all defined in the **<hw/sysinfo.h>** file.

The *hwinfo* section doesn't have to describe *all* the hardware. For instance, the startup program doesn't have to do PCI queries to discover what's been plugged into any slots if it doesn't want to. It's up to you as the startup implementor to decide how full to make the *hwinfo* description. As a rule, if a component is hardwired on your board, consider putting it into *hwinfo*.

Tags

Each structure (or *tag*) in the section starts the same way:

```
struct hwi_prefix {
    uint16_t        size;
    uint16_t        name;
};
```

The *size* field gives the size, in 4-byte quantities, of the structure (including the *hwi_prefix*).

The *name* field is an offset into the *strings* section of the system page, giving a zero-terminated string name for the structure. It might seem wasteful to use an ASCII

string rather than an enumerated type to identify the structure, but it actually isn't. The system page is typically allocated in 4 KB granularity, so the extra storage required by the strings doesn't cost anything. On the upside, people can add new structures to the section without requiring QNX Software Systems to act as a central repository for handing out enumerated type values. When processing the section, code should ignore any tag that it doesn't recognize (using the *size* field to skip over it).

Items

Each piece of hardware is described by a sequence of tags. This conglomeration of tags is known as an *item*. Each item describes one piece of hardware. The first tag in each item always starts out with the following structure (note that the first thing in it is a *hwi_prefix* structure):

```
struct hwi_item {
    struct hwi_prefix   prefix;
    uint16_t            itemsize;
    uint16_t            itemname;
    uint16_t            owner;
    uint16_t            kids;
};
```

The *itemsize* field gives the distance, in 4-byte quantities, until the start of the next item tag.

The *itemname* gives an offset into the *strings* section of the system page for the name of the item being described. Note that this differs from the *prefix.name* field, which tells what type of the structure the *hwi_item* is buried in.

The *owner* field gives the offset, in bytes, from the start of the *hwinfo* section to the item that this item is owned by. This field allows groups of items to be organized in a tree structure, similar to a filesystem directory hierarchy. We'll see how this is used later. If the item is at the root of a tree of ownership, the *owner* field is set to HWI_NULL_OFF.

The *kids* field indicates how many other items call this one "daddy."

The code currently requires that the tag name of any item structure must start with an uppercase letter; nonitem tags have to start with a lowercase letter.

Device trees

The *hwinfo* section contains trees describing the various hardware devices on the board.

The general hierarchy for devices is:

/hw/*bus***/***devclass***/***device*

where:

hw the root of the hardware tree.

bus the bus the hardware is on (**pci**, **eisa**, etc.).

devclass the general class of the device (**serial**, **rtc**, etc.).

device the actual chip implementing the device (**8250**, **mc146818**, etc.).

Building the section

Two basic calls in the startup library are used to add things to the *hwinfo* section:

- *hwi_alloc_tag()*
- *hwi_alloc_item()*

```
void *hwi_alloc_tag(const char *name, unsigned size, unsigned align);
```

This call allocates a tag of size *size* with the tag name of *name*. If the structure contains any 64-bit integer fields within it, the *align* field should be set to **8**; otherwise, it should be **4**. The function returns a pointer to memory that can be filled in as appropriate. Note that the *hwi_prefix* fields are automatically filled in by the *hwi_alloc_tag()* function.

```
void *hwi_alloc_item(const char *name, unsigned size,
                     unsigned align, const char *itemname,
                     unsigned owner);
```

This call allocates an *item* structure. The first three parameters are the same as in the *hwi_alloc_tag()* function.

The *itemname* and *owner* parameters are used to set the *itemname* and *owner* fields of the *hwi_item* structure. All *hwi_alloc_tag()* calls done after a *hwi_alloc_item()* call are assumed to belong to that item and the *itemsize* field is adjusted appropriately.

Here are the general steps for building an item:

1. Call *hwi_alloc_item()* to build a top-level item (one with the owner field to be HWI_NULL_OFF).
2. Add whatever other tag structures you want in the item.
3. Use *hwi_alloc_item()* to start a new item. This item could be either another top-level one or a child of the first.

Note that you can build the items in any order you wish, provided that the parent is built *before* the child.

When building a child item, suppose you've remembered its owner in a variable or you know only its item name. In order to find out the correct value of the *owner* parameter, you can use the following function (which is defined in the C library, since it's useful for people processing the section):

```
unsigned hwi_find_item(unsigned start, ...);
```

The *start* parameter indicates where to start the search for the given item. For an initial call, it should be set to HWI_NULL_OFF. If the item found isn't the one wanted, then the return value from the first *hwi_find_item()* is used as the *start* parameter of the second call. The search will pick up where it left off. This can be repeated as many times as required (the return value from the second call going into the *start* parameter of the third, etc). The item being searched is identified by a sequence of *char* * parameters following *start*. The sequence is terminated by a NULL. The last string before the NULL is the bottom-level *itemname* being searched for, the string in front of that is the name of the item that owns the bottom-level item, etc.

For example, this call finds the first occurrence of an item called "foobar":

```
item_off = hwi_find_item(HWI_NULL_OFF, "foobar", NULL);
```

The following call finds the first occurrence of an item called "foobar" that's owned by "sam":

```
item_off = hwi_find_item(HWI_NULL_OFF, "sam", "foobar", NULL);
```

If the requested item can't be found, HWI_NULL_OFF is returned.

Other functions

The following functions are in the C library for use in processing the *hwinfo* section:

`unsigned hwi_tag2off(void *);`

Given a pointer to the start of a tag, return the offset, in bytes, from the beginning of the start of the *hwinfo* section.

`void *hwi_off2tag(unsigned);`

Given an offset, in bytes, from the start of the *hwinfo* section, return a pointer to the start of the tag.

`unsigned hwi_find_tag(unsigned` *start*`, int` *curr_item*`, const char *`*tagname*);

Find the tag named *tagname*. The *start* parameter works the same as the one in *hwi_find_item()*. If *curr_item* is nonzero, the search stops at the end of the current item (whatever item the *start* parameter points into). If *curr_item* is zero, the search continues until the end of the section. If the tag isn't found, HWI_NULL_OFF is returned.

Defaults

Before *main()* is invoked in the startup program, the library adds some initial entries to serve as a basis for later items.

HWI_TAG_INFO() is a macro defined in the **`<startup.h>`** header and expands out to the three *name, size, align* parameters for *hwi_alloc_tag()* and *hwi_alloc_item()* based on some clever macro names.

```
void
hwi_default() {
    hwi_tag     *tag;
    hwi_tag     *tag;

    hwi_alloc_item(HWI_TAG_INFO(group), HWI_ITEM_ROOT_AS,
                   HWI_NULL_OFF);
    tag = hwi_alloc_item(HWI_TAG_INFO(group), HWI_ITEM_ROOT_HW,
                         HWI_NULL_OFF);

    hwi_alloc_item(HWI_TAG_INFO(bus), HWI_ITEM_BUS_UNKNOWN,
                   hwi_tag2off(tag));

    loc = hwi_find_item(HWI_NULL_OFF, HWI_ITEM_ROOT_AS, NULL);

    tag = hwi_alloc_item(HWI_TAG_INFO(addrspace),
                         HWI_ITEM_AS_MEMORY, loc);
    tag->addrspace.base = 0;
    tag->addrspace.len  = (uint64_t)1 << 32;
    #ifndef __X86__
       loc = hwi_tag2off(tag);
    #endif
    tag = hwi_alloc_item(HWI_TAG_INFO(addrspace), HWI_ITEM_AS_IO,
                         loc);
    tag->addrspace.base = 0;
    #ifdef __X86__
        tag->addrspace.len  = (uint64_t)1 << 16;
    #else
        tag->addrspace.len  = (uint64_t)1 << 32;
    #endif
}
```

Predefined items and tags

These are the items defined in the **hw/sysinfo.h** file. Note that you're free to create additional items — these are just what we needed for our own purposes. You'll notice that all things are defined as **HWI_TAG_NAME_***, **HWI_TAG_ALIGN_***, and **struct hwi_***. The names are chosen that way so that the *HWI_TAG_INFO()* macro in startup works properly.

Group item

```
#define HWI_TAG_NAME_group  "Group"
#define HWI_TAG_ALIGN_group (sizeof(uint32_t))
struct hwi_group {
    struct hwi_item     item;
};
```

The Group item is used when you wish to group a number of items together. It serves the same purpose as a directory in a filesystem. For example, the *devclass* level of the **/hw** tree would use a Group item.

Bus item

```
#define HWI_TAG_NAME_bus    "Bus"
#define HWI_TAG_ALIGN_bus   (sizeof(uint32))
struct hwi_bus {
    struct hwi_item     item;
};
```

The Bus item tells the system about a hardware bus. Item names can be (but are not limited to):

```
#define HWI_ITEM_BUS_PCI        "pci"
#define HWI_ITEM_BUS_ISA        "isa"
#define HWI_ITEM_BUS_EISA       "eisa"
#define HWI_ITEM_BUS_MCA        "mca"
#define HWI_ITEM_BUS_PCMCIA     "pcmcia"
#define HWI_ITEM_BUS_UNKNOWN    "unknown"
```

Device item

```
#define HWI_TAG_NAME_device     "Device"
#define HWI_TAG_ALIGN_device    (sizeof(uint32))
struct hwi_device {
    struct hwi_item     item;
    uint32_t            pnpid;
};
```

The Device item tells the system about an individual device (the *device* level from the "Trees" section — the *devclass* level is done with a "Group" tag). The *pnpid* field is the Plug and Play device identifier assigned by Microsoft.

`location` tag

```
#define HWI_TAG_NAME_location   "location"
#define HWI_TAG_ALIGN_location  (sizeof(uint64))
struct hwi_location {
    struct hwi_prefix   prefix;
    uint32_t            len;
    uint64_t            base;
    uint16_t            regshift;
    uint16_t            addrspace;
};
```

Note that **`location`** is a simple tag, not an item. It gives the location of the hardware device's registers, whether in a separate I/O space or memory-mapped. There may be more than one of these tags in an item description if there's more than one grouping of registers.

The *base* field gives the physical address of the start of the registers. The *len* field gives the length, in bytes, of the registers. The *regshift* tells how much each register access is shifted by. If a register is documented at *offset* of a device, then the driver will actually access offset **`offset2^regshift`** to get to that register.

The *addrspace* field is an offset, in bytes, from the start of the *asinfo* section. It should identify either the **`memory`** or **`io`** address space item to tell whether the device registers are memory-mapped.

`irq` tag

```
#define HWI_TAG_NAME_irq        "irq"
#define HWI_TAG_ALIGN_irq       (sizeof(uint32))
struct hwi_irq {
    struct hwi_prefix   prefix;
    uint32_t            vector;
};
```

Note that this is a simple tag, not an item. The *vector* field gives the logical interrupt vector number of the device.

`diskgeometry` tag

```
#define HWI_TAG_NAME_diskgeometry   "diskgeometry"
#define HWI_TAG_ALIGN_diskgeometry  (sizeof(uint32))
struct hwi_diskgeometry {
    struct hwi_prefix   prefix;
    uint8_t             disknumber;
    uint8_t             sectorsize;   /* as a power of two */
    uint16_t            heads;
    uint16_t            cyls;
    uint16_t            sectors;
    uint32_t            nblocks;
};
```

Note that this is a simple tag, not an item. This is an x86-only mechanism used to transfer the information from the BIOS about disk geometry.

`pad` tag

```
#define HWI_TAG_NAME_pad        "pad"
#define HWI_TAG_ALIGN_pad       (sizeof(uint32))
struct hwi_pad {
    struct hwi_prefix   prefix;
};
```

Note that this is a simple tag, not an item. This tag is used when padding must be inserted to meet the alignment constraints for the subsequent tag.

cpuinfo

The *cpuinfo* area contains information about each CPU chip in the system, such as the CPU type, speed, capabilities, performance, and cache sizes. There are as many elements in the *cpuinfo* structure as the *num_cpu* member indicates (e.g. on a dual-processor system, there will be two *cpuinfo* entries).

This table is filled automatically by the library function *init_cpuinfo()*.

Member	Description
cpu	This is a number that represents the type of CPU. Note that this number will vary with the CPU architecture. For example, on the x86 processor family, this number will be the processor chip number (e.g. 386, 586). On MIPS and PowerPC, this is filled with the contents of the version registers.
speed	Contains the MHz rating of the processor. For example, on a 300 MHz MIPS R4000, this number would be 300.
flags	See below.
name	Contains an index into the *strings* member in the system page structure. The character string at the specified index contains an ASCII, NULL-terminated machine name (e.g. on a MIPS R4000 it will be the string "`R4000`").

continued...

Member	Description
ins_cache	Contains an index into the *cacheattr* array, described below. This index points to the first definition in a list for the *instruction* cache.
data_cache	Contains an index into the *cacheattr* array, described below. This index points to the first definition in a list for the *data* cache.

The *flags* member contains a bitmapped indication of the capabilities of the CPU chip. Note that the prefix for the manifest constant indicates which CPU family it applies to (e.g. PPC_ indicates this constant is for use by the PowerPC family of processors). In the case of no prefix, it indicates that it's generic to any CPU.

Here are the constants and their defined meanings:

This constant:	Means that the CPU has or supports:
CPU_FLAG_FPU	Floating Point Unit (FPU).
CPU_FLAG_MMU	Memory Management Unit (MMU), and the MMU is enabled (i.e. the CPU is currently in virtual addressing mode).
X86_CPU_CPUID	**CPUID** instruction.
X86_CPU_RDTSC	**RDTSC** instruction.
X86_CPU_INVLPG	**INVLPG** instruction.
X86_CPU_WP	**WP** bit in the **CR0** register.
X86_CPU_BSWAP	**BSWAP** instruction.
X86_CPU_MMX	MMX instructions.
X86_CPU_CMOV	**CMOV***xx* instructions.
X86_CPU_PSE	Page size extensions.
X86_CPU_PGE	TLB (Translation Lookaside Buffer) global mappings.
X86_CPU_MTRR	MTRR (Memory Type Range Register) registers.
X86_CPU_SEP	**SYSENTER/SYSEXIT** instructions.
X86_CPU_SIMD	SIMD instructions.
X86_CPU_FXSR	**FXSAVE/FXRSTOR** instructions.
X86_CPU_PAE	Extended addressing.

continued...

This constant:	Means that the CPU has or supports:
PPC_CPU_EAR	EAR (External Address Register) register.
PPC_CPU_HW_HT	Hardware hash table.
PPC_CPU_HW_POW	Power management.
PPC_CPU_FPREGS	Floating point registers.
PPC_CPU_SW_HT	Software hash table.
PPC_CPU_ALTIVEC	AltiVec extensions.
PPC_CPU_XAEN	Extended addressing.
PPC_CPU_SW_TLBSYNC	Sync TLBs.
PPC_CPU_TLB_SHADOW	Shadow registers in TLB handler.
PPC_CPU_DCBZ_NONCOHERENT	DCBZ problems.
PPC_CPU_STWCX_BUG	Requires a workaround to avoid a hardware problem with an unpaired **`stwcx.`** instruction when the kernel switches contexts.
MIPS_CPU_FLAG_PFNTOPSHIFT_MASK	Construct TLB entries.
MIPS_CPU_FLAG_MAX_PGSIZE_MASK	Maximum number of masks.
MIPS_CPU_FLAGS_MAX_PGSIZE_SHIFT	Maximum number of shifts.
MIPS_CPU_FLAG_L2_PAGE_CACHE_OPS	L2 cache.
MIPS_CPU_FLAG_64BIT	64-bit registers.
MIPS_CPU_FLAG_128BIT	128-bit registers.
MIPS_CPU_FLAG_SUPERVISOR	Supervisor mode.
MIPS_CPU_FLAG_NO_WIRED	No wired register.
MIPS_CPU_FLAG_NO_COUNT	No count register.

`syspage_entry` *cacheattr*

The *cacheattr* area contains information about the configuration of the on-chip and off-chip cache system. It also contains the *control()* callout used for cache control operations. This entry is filled by the library routines *init_cpuinfo()* and *init_cacheattr()*.

Note that *init_cpuinfo()* deals with caches implemented on the CPU itself; *init_cacheattr()* handles board-level caches.

Each entry in the *cacheattr* area consists of the following:

Member	Description
next	index to next lower level entry
line_size	size of cache line in bytes
num_lines	number of cache lines
flags	See below
control	callout supplied by startup code (see below).

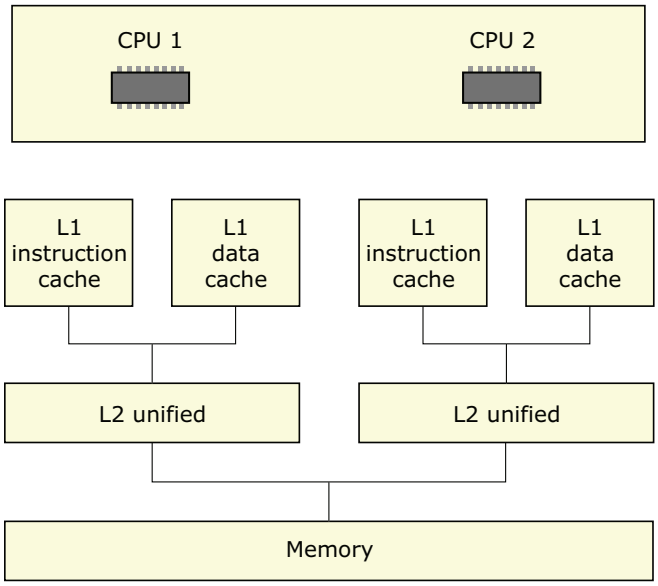
The total number of bytes described by a particular *cacheattr* entry is defined by *line_size* × *num_lines*.

The *flags* parameter is a bitmapped variable consisting of the following:

This constant:	Means that the cache:
CACHE_FLAG_INSTR	Holds instructions.
CACHE_FLAG_DATA	Holds data.
CACHE_FLAG_UNIFIED	Holds both instructions and data.
CACHE_FLAG_SHARED	Is shared between multiple processors in an SMP system.
CACHE_FLAG_SNOOPED	Implements a bus-snooping protocol.
CACHE_FLAG_VIRTUAL	Is virtually tagged.
CACHE_FLAG_WRITEBACK	Does write-back, not write-through.
CACHE_FLAG_CTRL_PHYS	Takes physical addresses via its *control()* function.
CACHE_FLAG_SUBSET	Obeys the *subset* property. This means that one cache level caches something from another level as well. As you go up each cache level, if something is in a particular level, it will also be in all the lower-level caches as well. This impacts the flushing operations of the cache in that a "subsetted" level can be effectively "ignored" by the *control()* function, since it knows that the operation will be performed on the lower-level cache.
CACHE_FLAG_NONCOHERENT	Is noncoherent on SMP.
CACHE_FLAG_NONISA	Doesn't obey ISA cache instructions.

The *cacheattr* entries are organized in a linked list, with the *next* member indicating the index of the next lower cache entry. This was done because some architectures will

have separate instruction and data caches at one level, but a unified cache at another level. This linking allows the system page to efficiently contain the information. Note that the entry into the *cacheattr* tables is done through the *cpuinfo*'s *ins_cache* and *data_cache*. Since the *cpuinfo* is an array indexed by the CPU number for SMP systems, it's possible to construct a description of caches for CPUs with different cache architectures. Here's a diagram showing a two-processor system, with separate L1 instruction and data caches as well as a unified L2 cache:



Two-processor system with separate L1 instruction and data caches.

Given the above memory layout, here's what the *cpuinfo* and *cacheattr* fields will look like:

```
/*
 * CPUINFO
 */
cpuinfo [0].ins_cache  = 0;
cpuinfo [0].data_cache = 1;

cpuinfo [1].ins_cache  = 0;
cpuinfo [1].data_cache = 1;

/*
 * CACHEATTR
 */
cacheattr [0].next = 2;
cacheattr [0].linesize = linesize;
cacheattr [0].numlines = numlines;
cacheattr [0].flags = CACHE_FLAG_INSTR;

cacheattr [1].next = 2;
cacheattr [1].linesize = linesize;
cacheattr [1].numlines = numlines;
cacheattr [1].flags = CACHE_FLAG_DATA;

cacheattr [2].next = CACHE_LIST_END;
```

```
cacheattr [2].linesize = linesize;
cacheattr [2].numlines = numlines;
cacheattr [2].flags = CACHE_FLAG_UNIFIED;
```

Note that the actual values chosen for *linesize* and *numlines* will, of course, depend on the actual configuration of the caches present on the system.

`syspage_entry` *qtime*

The *qtime* area contains information about the timebase present on the system, as well as other time-related information. The library routine *init_qtime()* fills these data structures.

Member	Description
intr	Contains the interrupt vector that the clock chip uses to interrupt the processor.
boot_time	Seconds since Jan 1 1970 00:00:00 GMT when the system was booted.
nsec	This 64-bit field holds the number of nanoseconds since the system was booted.
nsec_tod_adjust	When added to the *nsec* field, this field gives the number of nanoseconds from the start of the epoch (1970).
nsec_inc	Number of nanoseconds deemed to have elapsed each time the clock triggers an interrupt.
adjust	Set to zero at startup — contains any current timebase adjustment runtime parameters (as specified by the kernel call *ClockAdjust()*).
timer_rate	Used in conjunction with *timer_scale* (see below).
timer_scale	See below.
timer_load	Timer chip divisor value. The startup program leaves this zero. The kernel sets it based on the last *ClockPeriod()* and *timer_rate*/*timer_scale* values to a number, which is then put into the timer chip by the **`timer_load`**/**`timer_reload`** kernel callouts.
cycles_per_sec	For *ClockCycles()*.
epoch	Currently set to 1970, but not used.
flags	Indicates when timer hardware is specific to CPU0.

The *nsec* field is always monotonically increasing and is never affected by setting the current time of day via *ClockTime()* or *ClockAdjust()*. Since both *nsec* and *nsec_tod_adjust* are modified in the kernel's timer interrupt handler and are too big to load in an atomic read operation, to inspect them you must either:

- disable interrupts

  or:

- get the value(s) twice and make sure that they haven't changed between the first and second read.

The parameters *timer_rate* and *timer_scale* relate to the external counter chip's input frequency, in Hz, as follows:

$$\frac{1}{timer\_rate \times 10^{timer\_scale}}$$

Yes, this does imply that *timer_scale* is a negative number. The goal when expressing the relationship is to make *timer_rate* as large as possible in order to maximize the number of significant digits available during calculations.

For example, on an x86 PC with standard hardware, the values would be **`838095345UL`** for the *timer_rate* and **`-15`** for the *timer_scale*. This indicates that the timer value is specified in femtoseconds (the **`-15`** means "ten to the negative fifteen"); the actual value is 838,095,345 femtoseconds (approximately 838 nanoseconds).

If you need to change the number of *nsecs* that the OS adds to the time when a tick fires, you can manually adjust the *nsec_inc* value in **`SYSPAGE_ENTRY (qtime)`**.

The idea is to adjust for differences between the clock interval and the real expired time. The closer they become the less need there is for *ClockAdjust()* calls.

What you'll need to do is find out the physical address of the syspage. If it's already in *nsec_inc* you won't need to modify startup. If not, modify startup to put it there. Then use the *mmap_device_memory()* function to make the physical address of the syspage writable. That is, get the offset to the read-only page, and map a new block of memory to the address.

You could give *ClockAdjust()* a value of 0 for the number of ticks, to indicate that you want to make this adjustment "permanent". If you don't want to do that, you can give the *ClockAdjust()* function the maximum possible value for *tick_count*.

When you call and modify *nsec_inc*, you overwrite the *ClockPeriod()* function. The *timer_rate* and *timer_scale* fields are used as the input frequency to the clock hardware. The code uses these fields and the requested tick rate to calculate the number of input frequency clocks to count before generating an interrupt. The number of input frequency clocks that are counted, combined with *timer_rate* and *timer_scale* provides the *nsec_inc* value. For example:

```
timer_load = requested_ticksize / (timer_rate ** timer_scale)
```

```
nsec_inc   = timer_load * (timer_rate ** timer_scale)
```

The *nsec_inc* value is used to adjust the time of day when the clock interrupt goes off.

The changed value in *ClockPeriod()* is used to determine the new ticksize.

callout

The *callout* area is where various callouts get bound into. These callouts allow you to "hook into" the kernel and gain control when a given event occurs. The callouts operate in an environment similar to that of an interrupt service routine — you have a very limited stack, and you can't invoke any kernel calls (such as mutex operations, etc.). On standard hardware platforms (MIPS and PowerPC eval boards, x86-PC compatibles), you won't have to supply any functionality — it's already provided by the startup code we supply.

Member	Description
reboot	Used by the kernel to reset the system.
power	Provided for power management.
timer_load *timer_reload* *timer_value*	The kernel uses these *timer_** callouts to deal with the hardware timer chip.
debug	Used by the kernel when it wishes to interact with a serial port, console, or other device (e.g. when it needs to print out some internal debugging information or when there's a fault).

For details about the characteristics of the callouts, please see the sections "Callout information" and "Writing your own kernel callout" later in this chapter.

callin

For internal use.

typed_strings

The *typed_strings* area consists of several entries, each of which is a number and a string. The number is 4 bytes and the string is NULL-terminated as per C. The number in the entry corresponds to a particular constant from the system include file **<confname.h>** (see the C function *confname()* for more information).

Generally, you wouldn't access this member yourself; the various *init_*()* library functions put things into the typed strings literal pool themselves. But if you need to add something, you can use the function call *add_typed_string()* from the library.

strings

This member is a literal pool used for nontyped strings. Users of these strings would typically specify an index into *strings* (for example, *cpuinfo*'s *name* member).

Generally, you wouldn't access this member yourself; the various *init_*()* library functions put things into the literal pool themselves. But if you need to add something, you can use the function call *add_string()* from the library.

intrinfo

The *intrinfo* area is used to store information about the interrupt system. It also contains the callouts used to manipulate the interrupt controller hardware.

On a multicore system, each interrupt is directed to one (and only one) CPU, although it doesn't matter which. How this happens is under control of the programmable interrupt controller chip(s) on the board. When you initialize the PICs at startup, you can program them to deliver the interrupts to whichever CPU you want to; on some PICs you can even get the interrupt to rotate between the CPUs each time it goes off.

For the startups we write, we typically program things so that all interrupts (aside from the one(s) used for interprocessor interrupts) are sent to CPU 0. This lets us use the same startup for both **`procnto`** and **`procnto-smp`**. According to a study that Sun did a number of years ago, it's more efficient to direct all interrupts to one CPU, since you get better cache utilization.

The *intrinfo* area is automatically filled in by the library routine *init_intrinfo()*.

If you need to override some of the defaults provided by *init_intrinfo()*, or if the function isn't appropriate for your custom environment, you can call *add_interrupt_array()* directly with a table of the following format:

In all probability, you *will* need to modify this for non-x86 platforms.

Member	Description
vector_base	The base number of the logical interrupt numbers that programs will use (e.g. the interrupt vector passed to *InterruptAttach()*).
num_vectors	The number of vectors starting at *vector_base* described by this entry.
cascade_vector	If this interrupt entry describes a set of interrupts that are cascaded into another interrupt controller, then this variable contains the logical interrupt number that this controller cascades into.

continued...

Member	Description
cpu_intr_base	The association between this set of interrupts and the CPU's view of the source of the interrupt (see below).
cpu_intr_stride	The spacing between interrupt vector entries for interrupt systems that do autovectoring. On an x86 platform with the standard 8259 controller setup, this is the value 1, meaning that the interrupt vector corresponding to the hardware interrupt sources is offset by 1 (e.g. interrupt vector 0 goes to interrupt 0x30, interrupt vector 1 goes to interrupt 0x31, and so on). On non-x86 systems it's usually 0, because those interrupt systems generally don't do autovectoring. A value of 0 indicates that it's not autovectored.
flags	Used by the startup code when generating the kernel's interrupt service routine entry points. See below under INTR_FLAG_* and PPC_INTR_FLAG_*.
id	A code snippet that gets copied into the kernel's interrupt service routine used to identify the source of the interrupt, in case of multiple hardware events being able to trigger one CPU-visible interrupt. Further modified by the INTR_GENFLAG_* flags, defined below.
eoi	A code snippet that gets copied into the kernel's interrupt service routine that provides the *EOI* (End Of Interrupt) functionality. This code snippet is responsible for telling the controller that the interrupt is done and for unmasking the interrupt level. For CPU fault-as-an-interrupt handling, *eoi* identifies the cause of the fault.
mask	An outcall to mask an interrupt source at the hardware controller level. The numbers passed to this function are the interrupt vector numbers (starting at 0 to *num_vectors* - 1).
unmask	An outcall to unmask an interrupt source at the hardware controller level. Same vector numbers as *mask*, above.
config	Provides configuration information on individual interrupt levels. Passed the system page pointer (1st argument), a pointer to this interrupt info entry (2nd argument), and the zero-based interrupt level. Returns a bitmask; see INTR_CONFIG_FLAG* below.
patch_data	Provides information about patched data. The patched data is passed to the *patcher()* routine that gets called once for each callout in a *startup_intrinfo()* structure.

Each group of callouts (i.e. *id*, *eoi*, *mask*, *unmask*) for each level of interrupt controller deals with a set of interrupt vectors that start at 0 (zero-based). Set the callouts for each level of interruption accordingly.

Interrupt vector numbers are passed without offset to the callout routines. The association between the zero-based interrupt vectors the callouts use and the system-wide interrupt vectors is configured within the startup-intrinfo structures. These structures are found in the *init_intrinfo()* routine of startup.

The *cpu_intr_base* member

The interpretation of the *cpu_intr_base* member varies with the processor:

Processor	Interpretation
x86	The *IDT* (Interrupt Descriptor Table) entry, typically 0x30.
PPC	The offset from the beginning of the exception table where execution begins when an external interrupt occurs. A sample value is 0x0140, calculated by `0x0500 ÷ 4`.
PPC/BE	Interrupts no longer start at fixed locations in low memory. Instead there's a set of IVOR (Interrupt Vector Offset Register) registers. Each exception class has a different IVOR. When you specify the interrupt layout to startup, you'll need to identify the particular IVOR register the processor will use when the interrupt occurs. For example, PPCBKE_SPR_IVOR4 is used for normal external interrupts; PPCBKE_SPR_IVOR10 is used for decrementer interrupts. See `startup/boards/440rb/init_intrinfo.c` for an example of what to do on bookE CPUs.
PPC/Non-BE	—
MIPS	The value in the "cause" register when an external interrupt occurs. A sample value is 0.
ARM	This value should be 0, since all ARM interrupts are handled via the IRQ exception.
SH	The offset from the beginning of the exception table where execution starts when an interrupt occurs. For example, for 7750, the value is `0x600`.

The *flags* member

The *flags* member takes two sets of flags. The first set deals with the characteristics of the interrupts:

INTR_FLAG_NMI
: Indicates that this is a NonMaskable Interrupt (NMI). An NMI is an interrupt which can't be disabled by clearing the CPU's interrupt enable flag, unlike most normal interrupts. NonMaskable interrupts are typically used to signal events that require immediate action, such as a parity error, a hardware failure, or imminent loss of power. The address for the handler's NMI is stored in the BIOS's Interrupt Vector table at position 02H. For this reason an NMI is often referred to as INT 02H.

  The code in the kernel needs to differentiate between normal interrupts and NMIs, because with an NMI the kernel needs to know that it can't protect (mask) the interrupt (hence the "N" in NonMaskable Interrupt). We strongly discourage the use of the NMI vector in x86 designs; we don't support it on any non-x86 platforms.

Regular interrupts that are normally used and referred to by number are called maskable interrupts. Unlike non maskable interrupts, maskable interrupts are those that can be masked, or ignored, to allow the processor to complete a task.

INTR_FLAG_CASCADE_IMPLICIT_EOI
: Indicates that an EOI to the primary interrupt controller is not required when handling a cascaded interrupt (e.g. it's done automatically). Only used if this entry describes a cascaded controller.

INTR_FLAG_CPU_FAULT
: Indicates that one or more of the vectors described by this entry is *not* connected to a hardware interrupt source, but rather is generated as a result of a CPU fault (e.g. bus fault, parity error). Note that we strongly discourage designing your hardware this way. The implication is that a check needs to be inserted for an exception into the generated code stream; after the interrupt has been identified, an EOI needs to be sent to the controller. The EOI code burst has the additional responsibility of detecting what address caused the fault, retrieving the fault type, and then passing the fault on. The primary disadvantage of this approach is that it causes extra code to be inserted into the code path.

PPC_INTR_FLAG_400ALT

Similar to INTR_FLAG_NMI, this indicates to the code generator that a different kernel entry sequence is required. This is because the PPC400 series doesn't have an NMI, but rather has a critical interrupt that *can* be masked. This interrupt shows up differently from a "regular" external interrupt, so this flag indicates this fact to the kernel.

PPC_INTR_FLAG_CI

Same as PPC_INTR_FLAG_400ALT, where CI refers to critical interrupt.

PPC_INTR_FLAG_SHORTVEC

Indicates that exception table doesn't have normal 256 bytes of memory space between this and the next vector.

The second set of flags deals with code generation:

INTR_GENFLAG_LOAD_SYSPAGE

Before the interrupt identification or EOI code sequence is generated, a piece of code needs to be inserted to fetch the system page pointer into a register so that it's usable within the identification code sequence.

INTR_GENFLAG_LOAD_INTRINFO

Same as INTR_GENFLAG_LOAD_SYSPAGE, except that it loads a pointer to this structure.

INTR_GENFLAG_LOAD_INTRMASK

Used only by EOI routines for hardware that doesn't automatically mask at the chip level. When the EOI routine is about to reenable interrupts, it should reenable only those interrupts that are actually enabled at the user level (e.g. managed by the functions *InterruptMask()* and *InterruptUnmask()*). When this flag is set, the existing interrupt mask is stored in a register for access by the EOI routine. A zero in the register indicates that the interrupt should be unmasked; a nonzero indicates it should remain masked.

INTR_GENFLAG_NOGLITCH

Used by the interrupt ID code to cause a check to be made to see if the interrupt was due to a glitch or to a different controller. If this flag is set, the check is omitted — you're indicating that there's no reason (other than the fact that the hardware actually did generate an interrupt) to be in the interrupt service routine. If this flag is not set, the check is made to verify that the suspected hardware really is the source of the interrupt.

INTR_GENFLAG_LOAD_CPUNUM

Same as INTR_GENFLAG_LOAD_SYSPAGE, except that it loads a pointer to the number of the CPU this structure uses.

INTR_GENFLAG_ID_LOOP

Some interrupt controllers have read-and-clear registers indicating the active interrupts. That is, the first read returns a bitset with the pending interrupts, and then immediately zeroes the register. Since the interrupt ID callout can return only one interrupt number at a time, that means that we might fail to process all the interrupts if there's more than one bit on in the status register.

When INTR_GENFLAG_ID_LOOP is on, the kernel generates code to jump back to the ID callout after the EOI has finished.

In the ID callout, you need to allocate read-write storage as per the usual procedures. This storage is initially set to zero (done by default). When the callout runs, the first thing it does is check the storage area:

- If the storage is nonzero, the callout uses it to identify another interrupt to process, knocks that bit down, writes the new value back into the storage location and returns the identified interrupt number.
- If the storage location is zero, the callout reads the hardware status register (clearing it) and identifies the interrupt number from it. It then knocks that bit off, writes the value to the storage location, and then returns the appropriate interrupt number.
- If both the storage and hardware register are zero, the routine returns -1 to indicate no interrupt is present as per usual.

config return values

The *config* callout may return zero or more of the following flags:

INTR_CONFIG_FLAG_PREATTACH

Normally, an interrupt is masked off until a routine attaches to it via *InterruptAttach()* or *InterruptAttachEvent()*. If CPU fault indications are routed through to a hardware interrupt (*not* recommended!), the interrupt would, by default, be disabled. Setting this flag causes a "dummy" connection to be made to this source, causing this level to become unmasked.

INTR_CONFIG_FLAG_DISALLOWED

Prevents user code from attaching to this interrupt level. Generally used with INTR_CONFIG_FLAG_PREATTACH, but could be used to prevent user code from attaching to any interrupt in general.

INTR_CONFIG_FLAG_IPI

Identifies the vector that's used as the target of an inter-processor interrupt in an SMP system.

`syspage_entry` `union` *un*

The *un* union is where processor-specific system page information is kept. The purpose of the union is to serve as a demultiplexing point for the various CPU families. It is demultiplexed based on the value of the *type* member of the system page structure.

Member	Processor	*type*
x86	The x86 family	SYSPAGE_X86
ppc	PowerPC family	SYSPAGE_PPC
mips	The MIPS family	SYSPAGE_MIPS
arm	The ARM family	SYSPAGE_ARM
sh	The Hitachi SH family of processors.	SYSPAGE_SH

un.x86

This structure contains the x86-specific information. On a standard PC-compatible platform, the library routines (described later) fill these fields:

smpinfo Contains info on how to manipulate the SMP control hardware; filled in by the library call *init_smp()*.

gdt Contains the Global Descriptor Table (GDT); filled in by the library.

idt Contains the Interrupt Descriptor Table (IDT); filled in by the library.

pgdir Contains pointers to the Page Directory Table(s); filled in by the library.

real_addr The virtual address corresponding to the physical address range `0` through `0xFFFFF` inclusive (the bottom 1 megabyte).

un.x86.smpinfo (deprecated)

The members of this field are filled automatically by the function *init_smp()* within the startup library.

un.ppc (deprecated)

This structure contains the PowerPC-specific information. On a supported evaluation platform, the library routines (described later) fill these fields. On customized hardware, you'll have to supply the information.

smpinfo Contains info on how to manipulate the SMP control hardware; filled in by the library call *init_smp()*.

kerinfo Kernel information, filled by the library.

exceptptr Points at system exception table, filled by the library.

un.ppc.kerinfo

Contains information relevant to the kernel:

pretend_cpu Allows us to specify an override for the CPU ID register so that the kernel can pretend it is a "known" CPU type. This is done because the kernel "knows" only about certain types of PPC CPUs; different variants require specialized support. When a new variant is manufactured, the kernel will not recognize it. By stuffing the *pretend_cpu* field with a CPU ID from a known CPU, the kernel will pretend that it's running on the known variant.

init_msr Template of what bits to have on in the MSR when creating a thread. Since the MSR changes among the variants in the PPC family, this allows you to specify some additional bits that the kernel doesn't necessarily know about.

ppc_family Indicates what family the PPC CPU belongs to.

asid_bits Identifies what address space bits are active.

callout_ts_clear Lets callouts know whether to turn off data translation to get at their hardware.

un.mips

This structure contains the MIPS-specific information:

shadow_imask A shadow copy of the interrupt mask bits for the builtin MIPS interrupt controller.

un.arm

This structure contains the ARM-specific information:

L1_vaddr Virtual address of the MMU level 1 page table used to map the kernel.

L1_paddr Physical address of the MMU level 1 page table used to map the kernel.

startup_base Virtual address of a 1-1 virtual-physical mapping used to map the startup code that enables the MMU. This virtual mapping is removed when the kernel is initialized.

startup_size Size of the mapping used for *startup_base*.

cpu Structure containing ARM core-specific operations and data. Currently this contains the following:

- *page_flush* — A routine used to implement CPU-specific cache/TLB flushing when the memory manager unmaps or changes the access protections to a virtual memory mapping for a page. This routine is called for each page in a range being modified by the virtual memory manager.
- *page_flush_deferred* — A routine used to perform any operations that can be deferred when *page_flush* is called. For example on the SA-1110 processor, an *Icache* flush is deferred until all pages being operated on have been modified.

un.sh

This structure contains the Hitachi SH-specific information:

exceptptr Points at system exception table, filled by the library.

smp

The *smp* area is CPU-independent and contains the following elements:

This element	Description
send_ipi	Sends an interprocess interrupt (IPI) to the CPU.
start_address	Get the starting address for the IPI.
pending	Identify the pending interrupts for the SMP processor.
cpu	Identify the SMP CPU.

pminfo

The *pminfo* area is a communication area between the power manager and startup/power callout.

The *pminfo* area contains the following elements which are customizable in the power manager structure and are power-manager dependent:

This element	Description
wakeup_pending	Notifies the power callout that a wakeup condition has occurred. The power manager requires write access so it can modify this entry.
wakeup_condition	Indicates to the power manager what has caused the wakeup i.e. whether it's a power-on reset, or an interrupt from peripherals or other devices. The value is set by the power callout.
managed_storage	This entry is an area where the power manager can store any data it chooses. This storage is not persistent storage; it needs to be manually stored and restored by the startup and power callout. The *managed_storage* element is initialized by the *init_pminfo()* function call in startup and can be modified at startup. The value passed into *init_pminfo()* determines the size of the *managed_storage* array.

Callout information

All the callout routines share a set of similar characteristics:

- coded in assembler
- position-independent
- no static read/write storage

Callouts are basically binding standalone pieces of code for the kernel to invoke without having to statically link them to the kernel.

The requirement for coding the callouts in assembler stems from the second requirement (i.e. that they must be written to be position-independent). This is because the callouts are provided as part of the startup code, which will get overwritten when the kernel starts up. In order to circumvent this, the startup program will copy the callouts to a safe place — since they won't be in the location that they were loaded in, they must be coded to be position-independent.

We need to qualify the last requirement (i.e. that callouts not use any static read/write storage). There's a mechanism available for a given callout to allocate a small amount of storage space within the system page, but the callouts cannot have any static read/write storage space defined within themselves.

Debug interface

The debug interface consists of the following callouts:

- *display_char()*
- *poll_key()*
- *break_detect()*.

These three callouts are used by the kernel when it wishes to interact with a serial port, console, or other device (e.g. when it needs to print out some internal debugging information or when there's a fault). Only the *display_char()* is required; the others are optional.

Clock/timer interface

Here are the clock and timer interface callouts:

- *timer_load()*
- *timer_reload()*
- *timer_value()*.

The kernel uses these callouts to deal with the hardware timer chip.

The *timer_load()* callout is responsible for stuffing the divisor value passed by the kernel into the timer/counter chip. Since the kernel doesn't know the characteristics of the timer chip, it's up to the *timer_load()* callout to take the passed value and validate it. The kernel will then use the new value in any internal calculations it performs. You can access the new value in the *qtime_entry* element of the system page as well as through the *ClockPeriod()* function call.

The *timer_reload()* callout is called after the timer chip generates an interrupt. It's used in two cases:

- Reloading the divisor value (because some timer hardware doesn't have an automatic reload on the timer chip — this type of hardware should be avoided if possible).
- Telling the kernel whether the timer chip caused the interrupt or not (e.g. if you had multiple interrupt sources tied to the same line used by the timer — not the ideal hardware design, but. . .).

The *timer_value()* callout is used to return the value of the timer chip's internal count as a delta from the last interrupt. This is used on processors that don't have a high-precision counter built into the CPU (e.g. 80386, 80486).

Interrupt controller interface

Here are the callouts for the interrupt controller interface:

- *mask()*
- *unmask()*
- *config()*

In addition, two "code stubs" are provided:

- **`id`**

- **`eoi`**

The *mask()* and *unmask()* perform masking and unmasking of a particular interrupt vector.

The *config()* callout is used to ascertain the configuration of an interrupt level.

For more information about these callouts, refer to the *intrinfo* structure in the system page above.

Cache controller interface

Depending on the cache controller circuitry in your system, you may need to provide a callout for the kernel to interface to the cache controller.

On the x86 architecture, the cache controller is integrated tightly with the CPU, meaning that the kernel doesn't have to talk to the cache controller. On other architectures, like the MIPS and PowerPC, the cache controllers need to be told to invalidate portions of the cache when certain functions are performed in the kernel.

The callout for cache control is *control()*. This callout gets passed:

- a set of flags (defining the operation to perform)
- the address (either in virtual or physical mode, depending on flags in the *cacheattr* array in the system page)
- the number of cache lines to affect

The callout is responsible for returning the number of cache lines that it affected — this allows the caller (the kernel) to call the *control()* callout repeatedly at a higher level. A return of 0 indicates that the entire cache was affected (e.g. all cache entries were invalidated).

System reset callout

The miscellaneous callout, *reboot()*, gets called whenever the kernel needs to reboot the machine.

The *reboot()* callout is responsible for resetting the system. This callout lets developers customize the events that occur when proc needs to reboot — such as turning off a watchdog, banging the right registers etc. without customizing proc each time.

A "shutdown" of the binary will call *sysmgr_reboot()*, which will eventually trigger the *reboot()* callout.

Power management callout

The *power()* callout gets called whenever power management needs to be activated.

The *power()* callout is used for power management.

The startup library

The startup library contains a rich set of routines consisting of high-level functions that are called by your *main()* through to utility functions for interrogating the hardware, initializing the system page, loading the next process in the image, and switching to protected mode. Full source is provided for all these functions, allowing you to make local copies with minor modifications in your target startup directory.

The following are the available library functions (in alphabetical order):

add_cache()

```
int add_cache(int next,
              unsigned flags,
              unsigned line_size,
              unsigned num_lines,
              const struct callout_rtn *rtn);
```

Add an entry to the *cacheattr* section of the system page structure. Parameters map one-to-one with the structure's fields. The return value is the array index number of the added entry. Note that if there's already an entry that matches the one you're trying to add, that entry's index is returned — nothing new is added to the section.

add_callout()

```
void add_callout(unsigned offset,
                 const struct callout_rtn *callout);
```

Add a callout to the *callout_info* section of the system page. The *offset* parameter holds the offset from the start of the section (as returned by the *offsetof()* macro) that the new routine's address should be placed in.

add_callout_array()

```
void add_callout_array (const struct callout_slot *slots,
                        unsigned size)
```

Add the callout array specified by *slots* (for *size* bytes) into the callout array in the system page.

add_interrupt()

```
struct intrinfo_entry
        *add_interrupt(const struct startup_intrinfo
                                        *startup_intr);
```

Add a new entry to the *intrinfo* section. Returns a pointer to the newly added entry.

add_interrupt_array()

```
void add_interrupt_array (const struct startup_intrinfo *intrs,
                          unsigned size)
```

Add the interrupt array callouts specified by *intrs* (for *size* bytes) into the interrupt callout array in the system page.

add_ram()

```
void add_ram(paddr_t start,
             paddr_t size);
```

Tell the system that there's RAM available starting at physical address *start* for *size* bytes.

add_string()

```
unsigned add_string (const char *name)
```

Add the string specified by *name* into the string literal pool in the system page and return the index.

add_typed_string()

```
unsigned add_typed_string (int type_index,
                           const char *name)
```

Add the typed string specified by *name* (of type *type_index*) into the typed string literal pool in the system page and return the index.

alloc_qtime()

```
struct qtime_entry *alloc_qtime(void);
```

Allocate space in the system page for the *qtime* section and fill in the *epoch*, *boot_time*, and *nsec_tod_adjust* fields. Returns a pointer to the newly allocated structure so that user code can fill in the other fields.

alloc_ram()

```
paddr_t alloc_ram (paddr_t addr,
                   paddr_t size,
                   paddr_t align)
```

Allocate memory from the free memory pool initialized by the call to *init_raminfo()*. The RAM is *not* cleared.

as_add()

```
unsigned as_add(paddr_t start,
                paddr_t end,
                unsigned attr,
                const char *name,
                unsigned owner);
```

Add an entry to the *asinfo* section of the system page. Parameters map one-to-one with field names. Returns the offset from the start of the section for the new entry.

For more information and an example, see "Typed memory" in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

as_add_containing()

```
unsigned as_add_containing(paddr_t start,
                           paddr_t end,
                           unsigned attr,
                           const char *name,
                           const char *container);
```

Add new entries to the *asinfo* section, with the owner field set to whatever entries are named by the string pointed to by *container*. This function can add multiple entries because the *start* and *end* values are constrained to stay within the *start* and *end* of the containing entry (e.g. they get clipped such that they don't go outside the parent). If more than one entry is added, the AS_ATTR_CONTINUED bit will be turned on in all but the last. Returns the offset from the start of the section for the first entry added.

For more information and an example, see "Typed memory" in the Interprocess Communication (IPC) chapter of the *System Architecture* guide.

as_default()

```
unsigned as_default(void);
```

Add the default *memory* and *io* entries to the *asinfo* section of the system page.

as_find()

```
unsigned as_find(unsigned start, ...);
```

The *start* parameter indicates where to start the search for the given item. For an initial call, it should be set to AS_NULL_OFF. If the item found isn't the one wanted, then the return value from the first *as_find_item()* is used as the *start* parameter of the second call. The search will pick up where it left off. This can be repeated as many times as required (the return value from the second call going into the *start* parameter of the third, etc). The item being searched is identified by a sequence of *char* * parameters following *start*. The sequence is terminated by a NULL. The last string before the NULL is the bottom-level *itemname* being searched for, the string in front of that is the name of the item that owns the bottom-level item, etc.

For example, this call finds the first occurrence of an item called "foobar":

```
item_off = as_find_item(AS_NULL_OFF, "foobar", NULL);
```

The following call finds the first occurrence of an item called "foobar" that's owned by "sam":

```
item_off = as_find_item(AS_NULL_OFF, "sam", "foobar", NULL);
```

If the requested item can't be found, AS_NULL_OFF is returned.

as_find_containing()

```
unsigned as_find_containing(unsigned off,
                            paddr_t start,
                            paddr_t end,
                            const char *container);
```

Find an *asinfo* entry with the name pointed to by *container* that at least partially covers the range given by *start* and *end*. Follows the same rules as *as_find()* to know where the search starts. Returns the offset of the matching entry or AS_NULL_OFF if none is found. (The *as_add_containing()* function uses this to find what the owner fields should be for the entries it's adding.)

as_info2off()

```
unsigned as_info2off(const struct asinfo_entry *);
```

Given a pointer to an *asinfo* entry, return the offset from the start of the section.

as_off2info()

```
struct asinfo_entry *as_off2info(unsigned offset);
```

Given an offset from the start of the *asinfo* section, return a pointer to the entry.

as_set_checker()

```
void as_set_checker(unsigned off,
                    const struct callout_rtn *rtn);
```

Set the *checker* callout field of the indicated *asinfo* entry. If the AS_ATTR_CONTINUED bit is on in the entry, advance to the next entry in the section and set its priority as well (see *as_add_containing()* for why AS_ATTR_CONTINUED would be on). Repeat until an entry without AS_ATTR_CONTINUED is found.

as_set_priority()

```
void as_set_priority(unsigned as_off,
                     unsigned priority);
```

Set the *priority* field of the indicated entry. If the AS_ATTR_CONTINUED bit is on in the entry, advance to the next entry in the section and set its priority as well (see *as_add_containing()* for why AS_ATTR_CONTINUED would be on). Repeat until an entry without AS_ATTR_CONTINUED is found.

avoid_ram()

```
void avoid_ram( paddr32_t start,
                size_t size);
```

Make startup avoid using the specified RAM for any of its internal allocations. Memory remains available for **procnto** to use. This function is useful for specifying RAM that the IPL/ROM monitor needs to keep intact while startup runs. Because it takes only a **paddr32_t**, addresses can be specified in the first 4 GB. It doesn't need a full **paddr_t** because startup will never use memory above 4 GB for its own storage requirements.

calc_time_t()

```
unsigned long calc_time_t(const struct tm *tm);
```

Given a **struct tm** (with values appropriate for the UTC timezone), calculate the value to be placed in the *boot_time* field of the *qtime* section.

calloc_ram()

```
paddr32_t calloc_ram (size_t size,
                      unsigned align)
```

Allocate memory from the free memory pool initialized by the call to *init_raminfo()*. The RAM is cleared.

callout_io_map_indirect()

```
uintptr_t callout_io_map_indirect(unsigned size,
                                  paddr_t phys);
```

Same as *mmap_device_io()* in the C library — provide access to an I/O port on the x86 (for other systems, *callout_io_map()* is the same as *callout_memory_map_indirect()*) at a given physical address for a given size. The return value is for use in the CPU's equivalent of in/out instructions (regular moves on all but the x86). The value is for use in any kernel callouts (i.e. they live beyond the end of the startup program and are maintained by the OS while running).

callout_memory_map_indirect()

```
void *callout_memory_map_indirect(unsigned size,
                                  paddr_t phys,
                                  unsigned prot_flags);
```

Same as *mmap_device_memory()* in the C library — provide access to a memory-mapped device. The value is for use in any kernel callouts (i.e. they live beyond the end of the startup program and are maintained by the OS while running).

callout_register_data()

```
void callout_register_data( void *rp,
                            void *data );
```

This function lets you associate a pointer to arbitrary data with a callout. This data pointer is passed to the patcher routine (see "Patching the callout code," below.

The *rp* argument is a pointer to the pointer where the callout address is stored in the system page you're building. For example, say you have a pointer to a system page section that you're working on called *foo*. In the section there's a field *bar* that points to a callout when the system page is finished. Here's the code:

```
// This sets the callout in the syspage:

foo->bar = (void *)&callout_routine_name;

// This registers data to pass to the patcher when we're
// building the final version of the system page:

callout_register_data(&foo->bar, &some_interesting_data_for_patcher);
```

When the patcher is called to fix up the callout that's pointed at by **`foo->bar`**, &*some_interesting_data_for_patcher* is passed to it.

chip_access()

```
void chip_access(paddr_t base,
                 unsigned reg_shift,
                 unsigned mem_mapped,
                 unsigned size);
```

Get access to a hardware chip at physical address *base* with a register shift value of *reg_shift* (**`0`** if registers are one byte apart; **`1`** if registers are two bytes apart, etc. See **`devc-ser8250`** for more information).

If *mem_mapped* is zero, the function uses *startup_io_map()* to get access; otherwise, it uses *startup_memory_map()*. The *size* parameter gives the range of locations to be

given access to (the value is scaled by the *reg_shift* parameter for the actual amount that's mapped). After this call is made, the *chip_read*()* and *chip_write*()* functions can access the specified device. You can have only one *chip_access()* in effect at any one time.

chip_done()

```
void chip_done(void);
```

Terminate access to the hardware chip specified by *chip_access()*.

chip_read8()

```
unsigned chip_read8(unsigned off);
```

Read one byte from the device specified by *chip_access()*. The *off* parameter is first scaled by the *reg_shift* value specified in *chip_access()* before being used.

chip_read16()

```
unsigned chip_read16(unsigned off);
```

Same as *chip_read8()*, but for 16 bits.

chip_read32()

```
unsigned chip_read32(unsigned off);
```

Same as *chip_read16()*, but for 32 bits.

chip_write8()

```
void chip_write8(unsigned off,
                 unsigned val);
```

Write one byte from the device specified by *chip_access()*. The *off* parameter is first scaled by the *reg_shift* value specified in *chip_access()* before being used.

chip_write16()

```
void chip_write16(unsigned off,
                  unsigned val);
```

Same as *chip_write8()*, but for 16 bits.

chip_write32()

```
void chip_write32(unsigned off,
                  unsigned val);
```

Same as *chip_write16()*, but for 32 bits.

copy_memory()

```
void copy_memory (paddr_t dst,
                  paddr_t src,
                  paddr_t len)
```

Copy *len* bytes of memory from physical memory at *src* to *dst*.

del_typed_string()

```
int del_typed_string(int type_index);
```

Find the string in the *typed_strings* section of the system page indicated by the type *type_index* and remove it. Returns the offset where the removed string was, or -1 if no such string was present.

falcon_init_l2_cache()

```
void falcon_init_l2_cache(paddr_t base);
```

Enable the L2 cache on a board with a Falcon system controller chip. The base physical address of the Falcon controller registers are given by *base*.

falcon_init_raminfo()

```
void falcon_init_raminfo(paddr_t falcon_base);
```

On a system with the Falcon system controller chip located at *falcon_base*, determine how much/where RAM is installed and call *add_ram()* with the appropriate parameters.

falcon_system_clock()

```
unsigned falcon_system_clock(paddr_t falcon_base);
```

On a system with a Falcon chipset located at physical address *falcon_base*, return the speed of the main clock input to the CPU (in Hertz). This can then be used in turn to set the *cpu_freq*, *timer_freq*, and *cycles_freq* variables.

find_startup_info()

```
const void *find_startup_info (const void *start,
                               unsigned type)
```

Attempt to locate the kind of information specified by *type* in the data area used by the IPL code to communicate such information. Pass *start* as NULL to find the first occurrence of the given type of information. Pass *start* as the return value from a previous call in order to get the next information of that type. Returns 0 if no information of that type is found starting from *start*.

find_typed_string()

```
int find_typed_string(int type_index);
```

Return the offset from the beginning of the *type_strings* section of the string with the *type_index* type. Return -1 if no such string is present.

handle_common_option()

```
void handle_common_option (int opt)
```

Take the option identified by *opt* (a single ASCII character) and process it. This function assumes that the global variable *optarg* points to the argument string for the option.

Valid values for *opt* and their actions are:

A Reboot switch. If set, an OS crash will cause the system to reboot. If not set, an OS crash will cause the system to hang.

D Output channel specification (e.g. *kprintf()*, *stdout*, etc.).

f [*cpu_freq*][,[*cycles_freq*][,*timer_freq*]]

Specify CPU frequencies. All frequencies can be followed by **H** for hertz, **K** for kilohertz, or **M** for megahertz (these suffixes aren't case-sensitive). If no suffix is given, the library assumes megahertz if the number is less than 1000; otherwise, it assumes hertz.

If they're specified, *cpu_freq*, *cycles_freq*, and *timer_freq* are used to set the corresponding variables in the startup code:

cpu_freq — the CPU clock frequency. Also sets the speed field in the *cpuinfo* section of the system page.
cycles_freq — the frequency at which the value returned by *ClockCycles()* increments. Also sets the *cycles_per_sec* field in the *qtime* section of the system page.
timer_freq — the frequency at which the timer chip input runs. Also sets the *timer_rate* and *timer_scale* values of the *qtime* section of the system page.

K **kdebug** remote debug protocol channel.

M Placeholder for processing additional memory blocks. The parsing of additional memory blocks is deferred until *init_system_private()*.

N Add the hostname specified to the typed name string space under the identifier _CS_HOSTNAME.

R Used for reserving memory at the bottom of the address space.

r Used for reserving memory at any address space you specify.

S Placeholder for processing debug code's **-S** option.

P Specify maximum number of CPUs in an SMP system.

j Add Jtag-related options. Reserves four bytes of memory at the specified location and copies the *physical* address of the system page to this location so the hardware debugger can retrieve it.

v Increment the verbosity global flag, *debug_flag*.

hwi_add_device()

```
void hwi_add_device(const char *bus,
                    const char *class,
                    const char *name,
                    unsigned pnp);
```

Add an *hwi_device* item to the *hwinfo* section. The *bus* and *class* parameters are used to locate where in the device tree the new device is placed.

hwi_add_inputclk()

```
void hwi_add_inputclk(unsigned clk,
                      unsigned div);
```

Add an *hwi_inputclk* tag to the **hw** item currently being constructed.

hwi_add_irq()

```
void hwi_add_irq(unsigned vector);
```

Add an irq tag structure to the *hwinfo* section. The logical vector number for the interrupt will be set to *vector*.

hwi_add_location()

```
void hwi_add_location(paddr_t base,
                      paddr_t len,
                      unsigned reg_shift,
                      unsigned addr_space);
```

Add a location tag structure to the *hwinfo* section. The fields of the structure will be set to the given parameters.

hwi_add_nicaddr()

```
void hwi_add_nicaddr(const uint8 *addr,
                     unsigned len);
```

Add an *hwi_nicaddr* tag to the **hw** item currently being constructed.

hwi_add_rtc()

```
void hwi_add_rtc(const char *name,
                 paddr_t base,
                 unsigned reg_shift,
                 unsigned len,
                 int mmap,
                 int cent_reg);
```

Add an *hwi_device* item describing the realtime clock to the *hwinfo* section. The name of the device is *name*. The *hwi_location* tag items are given by *base*, *reg_shift*, *len*, and *mmap*. The *mmap* parameter indicates if the device is memory-mapped or I/O-space-mapped and is used to set the *addrspace* field.

If the *cent_reg* parameter is not -1, it's used to add an *hwi_regname* tag with the *offset* field set to its value. This indicates the offset from the start of the device where the century byte is stored.

hwi_alloc_item()

```
void *hwi_alloc_item(const char *tagname,
                     unsigned size,
                     unsigned align,
                     const char *itemname,
                     unsigned owner);
```

Add an item structure to the *hwinfo* section.

hwi_alloc_tag()

```
void *hwi_alloc_tag(const char *tagname,
                    unsigned size,
                    unsigned align);
```

Add a tag structure to the *hwinfo* section.

hwi_find_as()

```
unsigned hwi_find_as(paddr_t base,
int mmap);
```

Given a physical address of *base* and *mmap* (indicating 1 for memory-mapped and 0 for I/O-space-mapped), return the offset from the start of the *asinfo* section indicating the appropriate *addrspace* field value for an *hwi_location* tag.

hwi_find_item()

```
unsigned hwi_find_item(unsigned start, ...);
```

Although the *hwi_find_item()* function resides in the C library (proto in **<hw/sysinfo.h>**), the function is still usable from startup programs.

Search for a given item in the *hwinfo* section of the system page. If *start* is HWI_NULL_OFF, the search begins at the start of the *hwinfo* section. If not, it starts from the item after the offset of the one passed in (this allows people to find multiple tags of the same type; it works just like the *find_startup_info()* function). The *var args* portion is a list of character pointers, giving item names; the list is terminated with a NULL. The order of the item names gives ownership information. For example:

```
item = hwi_find_item(HWI_NULL_OFF, "foobar", NULL);
```

searches for an item name called “foobar.” The following:

```
item = hwi_find_item(HWI_NULL_OFF, "mumblyshwartz",
                     "foobar", NULL);
```

also searches for “foobar,” but this time it has to be owned by an item called “mumblyshwartz.”

If the item can’t be found, HWI_NULL_OFF is returned; otherwise, the byte offset within the *hwinfo* section is returned.

hwi_find_tag()

```
unsigned hwi_find_tag(unsigned start,
                      int curr_item,
                      const char *tagname);
```

Although the *hwi_find_tag()* function resides in the C library (proto in **<hw/sysinfo.h>**), the function is still usable from startup programs.

Search for a given tagname in the *hwinfo* section of startup. The *start* parameter works just like in *hwi_find_item()*. If *curr_item* is nonzero, the tagname must occur within the current item. If zero, the tagname can occur anywhere from the starting point of the search to the end of the section. If the tag can’t be found, then HWI_NULL_OFF is returned; otherwise, the byte offset within the *hwinfo* section is returned.

hwi_off2tag()

```
void *hwi_off2tag(unsigned off);
```

Although the *hwi_off2tag()* function resides in the C library (proto in **<hw/sysinfo.h>**), the function is still usable from startup programs.

Given a byte offset from the start of the *hwinfo* section, return a pointer to the *hwinfo* tag structure.

hwi_tag2off()

```
unsigned hwi_tag2off(void *tag);
```

Although the *hwi_tag2off()* function resides in the C library (proto in **<hw/sysinfo.h>**), the function is still usable from startup programs.

Given a pointer to the start of a *hwinfo* tag instruction, convert it to a byte offset from the start of the *hwinfo* system page section.

init_asinfo()

```
void init_asinfo(unsigned mem);
```

Initialize the *asinfo* section of the system page. The *mem* parameter is the offset of the **memory** entry in the section and can be used as the owner parameter value for *as_add()*s that are adding memory.

init_cacheattr()

```
void init_cacheattr (void)
```

Initialize the *cacheattr* member. For all platforms, this is a do-nothing stub.

init_cpuinfo()

```
void init_cpuinfo (void)
```

Initialize the members of the *cpuinfo* structure with information about the installed CPU(s) and related capabilities. Most systems will be able to use this function directly from the library.

init_hwinfo()

```
void init_hwinfo (void)
```

Initialize the appropriate variant of the *hwinfo* structure in the system page.

init_intrinfo()

```
void init_intrinfo (void)
```

Initialize the *intrinfo* structure.

x86 You would need to change this only if your hardware doesn't have the standard PC-compatible dual 8259 configuration.

MIPS	The default library version sets up the internal MIPS interrupt controller.
PowerPC	No default version exists; you must supply one.
ARM	No default version exists; you must supply one.
SH	The default library version sets up the SH-4 on-chip peripheral interrupt. You need to provide the external interrupt code.

If you're providing your own function, make sure it initializes:

- the interrupt controller hardware as appropriate (e.g. on the x86 it should program the two 8259 interrupt controllers)
- the *intrinfo* structure with the details of the interrupt controller hardware.

This initialization of the structure is done via a call to the function *add_interrupt_array()*.

init_mmu()

```
void init_mmu (void)
```

Sets up the processor for virtual addressing mode by setting up page-mapping hardware and enabling the pager.

On the x86 family, it sets up the page tables as well as special mappings to "known" physical address ranges (e.g. sets up a virtual address for the physical address ranges `0` through **`0xFFFFF`** inclusive).

The 400 and 800 series processors within the PowerPC family are stubs; the others, i.e. the 600 series and BookE processors, are not. On MIPS and SH, this function is currently a stub. On the PowerPC family, this function may be a stub.

On the ARM family, this function simply sets up the page tables.

init_pminfo()

```
*init_pminfo (unsigned managed_size)
```

Initialize the *pminfo* section of the system page and set the number of elements in the *managed storage* array.

init_qtime()

```
void init_qtime (void)
```

Initialize the *qtime* structure in the system page. Most systems will be able to use this function directly from the library.

This function doesn't exist for ARM. Specific functions exist for ARM processors with on-chip timers; currently, this includes only *init_qtime_sa1100()*.

init_qtime_sa1100()

```
void init_qtime_sa1100 (void)
```

Initialize the *qtime* structure and kernel callouts in the system page to use the on-chip timer for the SA1100 and SA1110 processors.

init_raminfo()

```
void init_raminfo (void)
```

Determine the location and size of available system RAM and initialize the *asinfo* structure in the system page.

If you know the exact amount and location of RAM in your system, you can replace this library function with one that simply hard-codes the values via one or more *add_ram()* calls.

x86	If the RAM configuration is known (e.g. set by the IPL code, or the multi-boot IPL code gets set by the gnu utility), then the library version of *init_raminfo()* will call the library routine *find_startup_info()* to fetch the information from a known location in memory. If the RAM configuration isn't known, then a RAM scan (via *x86_scanmem()*) is performed looking for valid memory between locations 0 and 0xFFFFFF, inclusive. (Note that the VGA aperture that usually starts at location 0xB0000 is specifically ignored.)
MIPS PowerPC ARM SH	There's no library default. You must supply your own *init_raminfo()* function.

init_smp()

```
void init_smp (void)
```

Initialize the SMP functionality of the system, assuming the hardware (e.g. x86, PPC, MIPS) supports SMP.

init_syspage_memory() (deprecated)

```
void init_syspage_memory (void *base,
                          unsigned size)
```

Initialize the system page structure's individual member pointers to point to the data areas for the system page substructures (e.g. *typed_strings*). The *base* parameter is a pointer to where the system page is currently stored (it will be moved to the kernel's address space later); the *size* indicates how big this area is. On all platforms, this routine shouldn't require modification.

init_system_private()

```
void init_system_private (void)
```

Find all the boot images that need to be started and fill a structure with that information; parse any **-M** options used to specify memory regions that should be added; tell Neutrino where the image filesystem is located; and finally allocate room for the actual storage of the system page. On all platforms, this shouldn't require modification.

Note that this must be the **last** *init_*()* function called.

jtag_reserve_memory()

```
void jtag_reserve_memory (unsigned long resmem_addr,
                          unsigned long resmem_size,
                          uint8_t resmem_flag)
```

Reserve a user-specified block of memory at the location specified in *resmem_addr*. If the *resmem_flag* is set to 0, clear the memory.

kprintf()

```
void kprintf (const char *fmt, ... )
```

Display output using the *put_char()* function you provide. It supports a very limited set of *printf()* style formats.

mips41xx_set_clock_freqs()

```
void mips41xx_set_clock_freqs(unsigned sysclk);
```

On a MIPS R41xx series chip, set the *cpu_freq*, *timer_freq*, and *cycles_freq* variables appropriately, given a system clock input frequency of *sysclk*.

openbios_init_raminfo()

```
void openbios_init_raminfo(void);
```

On a system that contains an OpenBIOS ROM monitor, add the system RAM information.

pcnet_reset()

```
void pcnet_reset(paddr_t base,
                 int mmap);
```

Ensure that a PCnet-style Ethernet controller chip at the given physical address (either I/O or memory-mapped as specified by *mmap*) is disabled. Some ROM monitors leave the Ethernet receiver enabled after downloading the OS image. This causes memory to be corrupted after the system starts and before Neutrino's Ethernet driver is run, due to the reception of broadcast packets. This function makes sure that no further packets are received by the chip until the Neutrino driver starts up and properly initializes it.

ppc400_pit_init_qtime()

```
void ppc400_pit_init_qtime(void);
```

On a PPC 400 series chip, initialize the *qtime* section and timer kernel callouts of the system page to use the on-board Programmable Interval Timer.

ppc405_set_clock_freqs()

```
void ppc405_set_clock_freqs
(unsigned sys_clk, unsigned timer_clk);
```

Initialize the *timer_freq* and *cycles_freq* variables based on a given *timer_clk*. The *cpu_freq* variable is initialized using a multiplication of a given system clock (*system_clk*). The multiplication value is found using the CPCO_PSR DCR.

ppc600_set_clock_freqs()

```
void ppc600_set_clock_freqs(unsigned sysclk);
```

On a PPC 600 series chip, set the *cpu_freq*, *timer_freq*, and *cycles_freq* variables appropriately, given a system clock input frequency of *sysclk*.

ppc700_init_l2_cache()

```
void ppc700_init_l2_cache(unsigned flags);
```

On a PPC 700 series system, initialize the L2 cache. The *flags* indicate which bits in the L2 configuration register are set. In particular, they decide the L2 size, clock speed, and so on. For details, see the Motorola PPC 700 series user's documentation for the particular hardware you're using.

For example, on a Sandpoint board, *flags* might be:

```
PPC700_SPR_L2CR_1M | PPC700_SPR_L2CR_CLK2 | PPC700_SPR_L2CR_OH05
```

This would set the following for L2CR:

- 1 MB L2 cache
- clock speed of half of the core speed
- "output-hold" value of 0.5 nsec.

ppc800_pit_init_qtime()

```
void ppc800_pit_init_qtime(void);
```

On a PPC 800 series chip, initialize the *qtime* section and timer kernel callouts of the system page to use the on-board Programmable Interval Timer.

ppc800_set_clock_freqs()

```
void ppc800_set_clock_freqs(unsigned extclk_freq,
                            unsigned extal_freq,
                            int is_extclk);
```

On a PPC 800 series chip, set the *cpu_freq*, *timer_freq*, and *cycles_freq* variables appropriately, given input frequencies of *extclk_freq* at the **EXTCLK** pin and *extal_freq* at the **XTAL/EXTAL** pins.

If *is_extclk* is nonzero, then the *extclk_freq* is used for the main timing reference (**MODCLK1** signal is one at reset). If zero, *extal_freq* is used at the main timing reference (**MODCLK1** signal is zero at reset).

Note that the setting of the frequency variables assumes that the *ppc800_pit_init_qtime()* routine is being used. If some other initialization of the *qtime* section and timer callouts takes place, the values in the frequency variables may have to be modified.

ppc_dec_init_qtime()

```
void ppc_dec_init_qtime(void);
```

On a PPC, initialize the *qtime* section and timer kernel callouts of the system page to use the decrementer register.

The *ppc_dec_init_qtime()* routine may not be used on a PPC 400 series chip, which omits the decrementer register.

print_syspage()

```
void print_syspage (void)
```

Print the contents of all the structures in the system page. The global variable *debug_level* is used to determine what gets printed. The *debug_level* must be at least 2 to print anything; a *debug_level* of 3 will print the information within the individual substructures.

Note that you can set the debug level at the command line by specifying multiple **-v** options to the startup program.

You can also use the startup program's **-s** command-line option to select which entries are printed from the system page: **-s***name* selects *name* to be printed, whereas **-s~***name* disables *name* from being printed. The *name* can be selected from the following list:

Name	Processors	Syspage entry
cacheattr	all	Cache attributes
callout	all	Callouts
cpuinfo	all	CPU info
gdt	x86	Global Descriptor Table
hwinfo	all	Hardware info
idt	x86	Interrupt Descriptor Table
intrinfo	all	Interrupt info

continued...

Name	Processors	Syspage entry
kerinfo	PPC	Kernel info
pgdir	x86	Page directory
qtime	all	System time info
smp	all	SMP info
strings	all	Strings
syspage	all	Entire system page
system_private	all	System private info
typed_strings	all	Typed strings

rtc_time()

```
unsigned long rtc_time (void)
```

This is a user-replaceable function responsible for returning the number of seconds since January 1 1970 00:00:00 GMT.

x86	This function defaults to calling *rtc_time_mc146818()*, which knows how to get the time from an IBM-PC standard clock chip.
MIPS PowerPC ARM	The default library version simply returns zero.
SH	The default function calls *rtc_time_sh4()*, which knows how to get the time from the SH-4 on-chip rtc.

Currently, these are the chip-specific versions:

rtc_time_ds1386()	Dallas Semiconductor DS-1386 compatible
rtc_time_m48t5x()	SGS-Thomson M48T59 RTC/NVRAM chip
rtc_time_mc146818()	Motorola 146818 compatible
rtc_time_rtc72423()	FOX RTC-72423 compatible
rtc_time_rtc8xx()	PPC 800 onboard RTC hardware

There's also a "none" version to use if your board doesn't have RTC hardware:

```
unsigned long rtc_time_none(void);
```

For the PPC 800 onboard RTC hardware, the function is simply as follows:

```
unsigned long rtc_time_rtc8xx(void);
```

If you're supplying the *rtc_time()* routine, you should call one of the chip-specific routines or write your own. The chip-specific routines all share the same parameter list:

```
(paddr_t base, unsigned reg_shift, int mmap, int cent_reg);
```

The *base* parameter indicates the physical base address or I/O port of the device. The *reg_shift* indicates the register offset as a power of two.

A typical value would be `0` (meaning 2^0, i.e. 1), indicating that the registers of the device are one byte apart in the address space. As another example, a value of `2` (meaning 2^2, i.e. 4) indicates that the registers in the device are four bytes apart.

If the *mmap* variable is `0`, then the device is in I/O space. If *mmap* is `1`, then the device is in memory space.

Finally, *cent_reg* indicates which register in the device contains the century byte (`-1` indicates no such register). If there's no century byte register, then the behavior is chip-specific. If the chip is year 2000-compliant, then we will get the correct time. If the chip isn't compliant, then if the year is less than 70, we assume it's in the range 2000 to 2069; else we assume it's in the range 1970 to 1999.

startup_io_map()

```
uintptr_t startup_io_map(unsigned size,
                         paddr_t phys);
```

Same as *mmap_device_io()* in the C library — provide access to an I/O port on the x86 (for other systems, *startup_io_map()* is the same as *startup_memory_map()*) at a given physical address for a given size. The return value is for use in the *in*/out** functions in the C library. The value is for use during the time the startup program is running (as opposed to *callout_io_map()*, which is for use after startup is completed).

startup_io_unmap()

```
void startup_io_unmap(uintptr_t port);
```

Same as *unmap_device_io()* in the C library — remove access to an I/O port on the x86 (on other systems, *unmap_device_io()* is the same as *startup_memory_unmap()*) at the given port location.

startup_memory_map()

```
void *startup_memory_map(unsigned size,
                         paddr_t phys,
                         unsigned prot_flags);
```

Same as *mmap_device_io_memory()* in the C library — provide access to a memory-mapped device. The value is for use during the time the startup program is running (as opposed to *callout_memory_map()*, which is for use after startup is completed).

startup_memory_unmap()

```
void startup_memory_unmap(void *vaddr);
```

Same as *unmap_device_memory()* in the C library — remove access to a memory-mapped device at the given location.

tulip_reset()

```
void tulip_reset(paddr_t phys,
                 int mem_mapped);
```

Ensure that a Tulip Ethernet chip (Digital 21x4x) at the given physical address (either I/O or memory-mapped as specified by *mem_mapped*) is disabled. Some ROM monitors leave the Ethernet receiver enabled after downloading the OS image. This causes memory to be corrupted after the system starts and before Neutrino's Ethernet driver is run, due to the reception of broadcast packets. This function makes sure that no further packets are received by the chip until the Neutrino driver starts up and properly initializes it.

uncompress()

```
int uncompress(char *dst,
               int *dstlen,
               char *src,
               int srclen,
               char *win);
```

This function resides in the startup library and is responsible for expanding a compressed OS image out to full size (this is invoked before *main()* gets called). If you know you're never going to be given a compressed image, you can replace this function with a stub version in your own code and thus make a smaller startup program.

x86_cpuid_string()

```
int x86_cpuid_string (char *buf,
                      int max)
```

Place a string representation of the CPU in the string *buf* to a maximum of *max* characters. The general format of the string is:

manufacturer part **F***family* **M***model* **S***stepping*

This information is determined using the **cpuid** instruction. If it's not supported, then a subset (typically only the *part*) will be placed in the buffer (e.g. 386).

x86_cputype()

```
unsigned x86_cputype (void)
```

An x86 platform-only function that determines the type of CPU and returns the number (e.g. 386).

x86_enable_a20()

```
int x86_enable_a20 (unsigned long cpu,
                    int only_keyboard)
```

Enable address line A20, which is often disabled on many PCs on reset. It first checks if address line A20 is enabled and if so returns 0. Otherwise, it sets bit **0x02** in port **0x92**, which is used by many systems as a fast A20 enable. It again checks to see if A20 is enabled and if so returns 0. Otherwise, it uses the keyboard microcontroller to enable A20 as defined by the old PC/AT standard. It again checks to see if A20 is enabled and if so returns 0. Otherwise, it returns -1.

If *cpu* is a 486 or greater, it issues a **wbinvd** opcode to invalidate the cache when doing a read/write test of memory to see if A20 is enabled.

In the rare case where setting bit **0x02** in port **0x92** may affect other hardware, you can skip this by setting *only_keyboard* to 1. In this case, it will attempt to use only the keyboard microcontroller.

x86_fputype()

```
unsigned x86_fputype (void)
```

An x86-only function that returns the FPU type number (e.g. 387).

x86_init_pcbios()

```
void x86_init_pcbios(void);
```

Perform initialization unique to an IBM PC BIOS system.

x86_pcbios_shadow_rom()

```
int x86_pcbios_shadow_rom(paddr_t rom,
                          size_t size);
```

Given the physical address of a ROM BIOS extension, this function makes a copy of the ROM in a RAM location and sets the x86 page tables in the **_syspage_ptr->un.x86.real_addr** range to refer to the RAM copy rather than the ROM version. When something runs in V86 mode, it'll use the RAM locations when accessing the memory.

The amount of ROM shadowed is the maximum of the *size* parameter and the size indicated by the third byte of the BIOS extension.

The function returns:

- **0** if there's no ROM BIOS extension signature at the address given
- **1** if you're starting the system in physical mode and there's no MMU to make a RAM copy be referenced
- **2** if everything works.

x86_scanmem()

```
unsigned x86_scanmem (paddr_t beg,
                      paddr_t end)
```

An x86-only function that scans memory between *beg* and *end* looking for RAM, and returns the total amount of RAM found. It scans memory performing a R/W test of 3 values at the start of each 4 KB page. Each page is marked with a unique value. It then rescans the memory looking for contiguous areas of memory and adds them to the *asinfo* entry in the system page.

A special check is made for a block of memory between addresses **0xB0000** and **0xBFFFF**, inclusive. If memory is found there, the block is skipped (since it's probably the dual-ported memory of a VGA card).

The call **x86_scanmem (0, 0xFFFFFF)** would locate all memory in the first 16 megabytes of memory (except VGA memory). You may make multiple calls to *x86_scanmem()* to different areas of memory in order to step over known areas of dual-ported memory with hardware.

Writing your own kernel callout

In order for the Neutrino microkernel to work on all boards, all hardware-dependent operations have been factored out of the code. Known as *kernel callouts*, these routines must be provided by the startup program.

The startup can actually have a number of different versions of the same callout available — during hardware discovery it can determine which one is appropriate for the board it's running on and make that particular instance of the callout available to the kernel. Alternatively, if you're on a deeply embedded system and the startup knows exactly what hardware is present, only one of each callout might be present; the startup program simply tells the kernel about them with no discovery process.

The callout code is copied from the startup program into the system page and after this, the startup memory (text and data) is freed.

At the point where the reboot callout is called:

- the MMU is enabled (the callout would have to disable it if necessary)
- you are running on the kernel stack
- you are executing code copied into the system page so no functions in the startup program are available.

The patch code is run during execution of the startup program itself, so regular calls work as normal.

Once copied, your code must be completely self-contained and position independent. The purpose of the patch routines is to allow you to patch up the code with constants, access to RW data storage etc. so that your code is self-contained and contains all the virtual-physical mappings required.

Find out who's gone before

The startup library provides a number of different callout routines that we've already written. You should check the source tree (originally installed in *bsp_working_dir*`/src/hardware/startup/lib/`) to see if a routine for your device/board is already available before embarking on the odyssey of writing your own. This directory includes generic code, as well as processor-specific directories.

In the CPU-dependent level of the tree for all the source files, look for files that match the pattern:

```
callout_*.[sS]
```

Those are all the callouts provided by the library. Whether a file ends in `.s` or `.S` depends on whether it's sent through the C preprocessor before being handed off to an assembler. For our purposes here, we'll simply refer to them as `.s` files.

The names break down further like this:

`callout_`*category_device*`.s`

where *category* is one of:

`cache`	cache control routines
`debug`	kernel debug input and output routines
`interrupt`	interrupt handling routines
`timer`	timer chip routine
`reboot`	rebooting the system

The *device* identifies the unique hardware that the callouts are for. Typically, all the routines in a particular source file would be used (or not) as a group by the kernel. For example, the `callout_debug_8250.s` file contains the *display_char_8250()*, *poll_key_8250()*, and *break_detect_8250()* routines for dealing with an 8250-style UART chip.

Why are they in assembly language?

Since the memory used by the startup executable is reclaimed by the OS after startup has finished, the callouts that are selected for use by the kernel can't be used in place. Instead, they must be copied to a safe location (the library takes care of this for you). Therefore, the callout code must be completely position-independent, which is why callouts have to be written in assembly language. We need to know where the callout begins and where it ends; there isn't a portable way to tell where a C function ends.

The other issue is that there isn't a portable way to control the preamble/postamble creation or code generation. So if an ABI change occurs or a build configuration issue occurs, we could have a very latent bug.

For all but two of the routines, the kernel invokes the callouts with the normal function-calling conventions. Later we'll deal with the two exceptions (*interrupt_id()* and *interrupt_eoi()*).

Starting off

Find a callout source file of the appropriate category that's close to what you want and copy it to a new filename. If the new routines will be useful on more than one board, you might want to keep the source file in your own private copy of the startup library. If not, you can just copy to the directory where you've put your board-specific files.

Now edit the new source file. At the top you'll see something that looks like this:

```
#include "callout.ah"
```

Or:

```
.include "callout.ah"
```

The difference depends on the assembler syntax being used.

This include file defines the CALLOUT_START and CALLOUT_END macros. The CALLOUT_START macro takes three parameters and marks the start of one callout. The first parameter is the name of the callout routine (we'll come back to the two remaining parameters later).

The CALLOUT_END macro indicates the end of the callout routine source. It takes one parameter, which has to be the same as the first parameter in the preceding CALLOUT_START. If this particular routine is selected to be used by the kernel, the startup library will copy the code between the CALLOUT_START and CALLOUT_END to a safe place for the kernel to use. The exact syntax of the two macros depends on exactly which assembler is being used on the source. Two common versions are:

```
CALLOUT_START(timer_load_8254, 0, 0)
CALLOUT_END(timer_load_8254)
```

Or:

```
CALLOUT_START timer_load_8254, 0, 0
CALLOUT_END timer_load_8254
```

Just keep whatever syntax is being used by the original file you started from. The original file will also have C prototypes for the routines as comments, so you'll know what parameters are being passed in. Now you should replace the code from the original file with what will work for the new device you're dealing with.

"Patching" the callout code

You may need to write a callout that deals with a device that may appear in different locations on different boards. You can do this by "patching" the callout code as it is copied to its final position. The third parameter of the CALLOUT_START macro is either a zero or the address of a *patcher()* routine. This routine has the following prototype:

```
void patcher(paddr_t paddr,
             paddr_t vaddr,
             unsigned rtn_offset,
             unsigned rw_offset,
             void *data,
             struct callout_rtn *src );
```

This routine is invoked immediately after the callout has been copied to its final resting place. The parameters are as follows:

paddr — Physical address of the start of the system page.

vaddr — Virtual address of the system page that allows read/write access (usable only by the kernel).

rtn_offset — Offset from the beginning of the system page to the start of the callout's code.

rw_offset — See the section on "Getting some R/W storage" below.

data — A pointer to arbitrary data registered by *callout_register_data()* (see above).

src — A pointer to the **callout_rtn** structure that's being copied into place.

The *data* and *src* arguments were added in the QNX Neutrino Core OS 6.3.2. Earlier patcher functions can ignore them.

Here's an example of a patcher routine for an x86 processor:

```
patch_debug_8250:
    movl    0x4(%esp),%eax                      // get paddr of routine
    addl    0xc(%esp),%eax                      // ...
        movl        0x14(%esp),%edx                         // get base info

    movl    DDI_BASE(%edx),%ecx         // patch code with real serial port
    movl    %ecx,0x1(%eax)
    movl    DDI_SHIFT(%edx),%ecx        // patch code with register shift
    movl    $REG_LS,%edx
    shll    %cl,%edx
    movl    %edx,0x6(%eax)
    ret


CALLOUT_START(display_char_8250, 0, patch_debug_8250)
    movl      $0x12345678,%edx          // get serial port base (patched)
    movl      $0x12345678,%ecx          // get serial port shift (patched)
....
CALLOUT_END(display_char_8250)
```

After the *display_char_8250()* routine has been copied, the *patch_debug_8250()* routine is invoked, where it modifies the constants in the first two instructions to the appropriate I/O port location and register spacing for the particular board. The patcher routines don't have to be written in assembler, but they typically are to keep them in

the same source file as the code they're patching. By arranging the first instructions in a group of related callouts all the same (e.g. *debug_char_*(), poll_key_*(), break_detect_*()*), the same patcher routine can be used for all of them.

Getting some R/W storage

Your callouts may need to have access to some static read/write storage. Normally this wouldn't be possible because of the position-independent requirements of a callout. But you can do it by using the patcher routines and the second parameter to CALLOUT_START. The second parameter to CALLOUT_START is the address of a four-byte variable that contains the amount of read/write storage the callout needs. For example:

```
rw_interrupt:
.long 4

patch_interrupt:
add a1,a1,a2
j ra
sh a3,0+LOW16(a1)

/*
 * Mask the specified interrupt
 */
CALLOUT_START(interrupt_mask_mips, rw_interrupt, patch_interrupt)
/*
 * Input Parameters :
 *      a0 - syspage_ptr
 *      a1 - Interrupt Number
 * Returns:
 * v0 - error status
 */

/*
 * Mark the interrupt disabled
 */
la  t3,0x1234(a0) # get enabled levels addr (patched)
li       t1, MIPS_SREG_IMASK0
....
CALLOUT_END(interrupt_mask_mips)
```

The *rw_interrupt* address as the second parameter tells the startup library that the routine needs four bytes of read/write storage (since the contents at that location is a **4**). The startup library allocates space at the end of the system page and passes the offset to it as the *rw_offset* parameter of the patcher routine. The patcher routine then modifies the initial instruction of the callout to the appropriate offset. While the callout is executing, the **t3** register will contain a pointer to the read/write storage. The question you're undoubtedly asking at this point is: *Why is the* CALLOUT_START *parameter the address of a location containing the amount of storage? Why not just pass the amount of storage directly?*

That's a fair question. It's all part of a clever plan. A group of related callouts may want to have access to shared storage so that they can pass information among themselves. The library passes the same *rw_offset* value to the patcher routine for all routines that share the same address as the second parameter to CALLOUT_START. In other words:

```
CALLOUT_START(interrupt_mask_mips, rw_interrupt, patch_interrupt)
....
CALLOUT_END(interrupt_mask_mips)

CALLOUT_START(interrupt_unmask_mips, rw_interrupt, patch_interrupt)
....
CALLOUT_END(interrupt_unmask_mips)

CALLOUT_START(interrupt_eoi_mips, rw_interrupt, patch_interrupt)
....
CALLOUT_END(interrupt_eoi_mips)

CALLOUT_START(interrupt_id_mips, rw_interrupt, patch_interrupt)
....
CALLOUT_END(interrupt_id_mips)
```

will all get the same *rw_offset* parameter value passed to *patch_interrupt()* and thus will share the same read/write storage.

The exception that proves the rule

To clean up a final point, the *interrupt_id()* and *interrupt_eoi()* routines aren't called as normal routines. Instead, for performance reasons, the kernel intermixes these routines directly with kernel code — the normal function-calling conventions aren't followed. The **`callout_interrupt_*.s`** files in the startup library will have a description of what registers are used to pass values into and out of these callouts for your particular CPU. Note also that you can't return from the middle of the routine as you normally would. Instead, you're required to "fall off the end" of the code.

PPC chips support

The PPC startup library has been modified in order to:

- minimize the number of locations that check the PVR SPR.
- minimize duplication of code.
- make it easier to leave out unneeded chip-dependent code.
- make it easier to add support for new CPUs.
- remove the notion of a PVR split into "family" and "member" fields.
- automatically take care of as much CPU-dependent code as possible in the library.

The new routines and data variables all begin with *ppcv_* for PPC variant, and are separated out into one function or data variable per source file. This separation allows maximum code reuse and minimum code duplication.

There are two new data structures:

- **`ppcv_chip`**
- **`ppcv_config`**

The first is:

```
struct ppcv_chip {
    unsigned short  chip;
    uint8_t         paddr_bits;
    uint8_t         cache_lsize;
    unsigned short  icache_lines;
    unsigned short  dcache_lines;
    unsigned        cpu_flags;
    unsigned        pretend_cpu;
    const char      *name;
    void            (*setup)(void);
};
```

Every supported CPU has a statically initialized variable of this type (in its own source file, e.g. **<ppvc_chip_603e7.c>**).

If the *chip* field matches the upper 16 bits of the PVR register, this **ppcv_chip** structure is selected and the *pccv* global variable in the library is pointed at it. Only the upper 16 bits are checked so you can use the constants like PPC_750 defined in **<ppc/cpu.h>** when initializing the field.

The *paddr_bits* field is the number of physical address lines on the chip, usually 32.

The *cache_lsize* field is the number of bits in a cache line size of the chip, usually 5, but sometimes 4.

The *icache_lines* and *dcache_lines* are the number of lines in the instruction and data cache, respectively.

The *cpu_flags* field holds the PPC_CPU_* flag constants from **<ppc/syspage.h>** that are appropriate for this CPU. Note that the older startups sometimes left out flags like PPC_CPU_HW_HT and depended on the kernel to check the PVR and turn them on if appropriate. This is no longer the case. The kernel will continue to turn on those bits if it detects an old style startup, but will NOT with a new style one.

The *pretend_cpu* field goes into the ppc_kerinfo_entry.pretend_cpu field of the system page and as before, it's used to tell the kernel that even though you don't know the PVR, you can act like it's the pretend one.

The *name* field is the string name of the CPU that gets put in the cpuinfo section.

The *setup* function is called when a particular ppcv_chip structure has been selected by the library as the one to use. It continues the library customization process by filling the second new structure.

The second data structure is:

```
struct ppcv_config {
    unsigned    family;
    void        (*cpuconfig1)(int cpu);
    void        (*cpuconfig2)(int cpu);
    void        (*cpuinfo)(struct cpuinfo_entry *cpu);
    void        (*qtime)(void);
    void        *(*map)(unsigned size, paddr_t phys,
                                unsigned prot_flags);
    void        (*unmap)(void *);
    int         (*mmu_info)(enum mmu_info info, unsigned tlb);
```

```
//NYI: tlb_read/write
};
```

There's a single variable defined of this type in the library, called *ppcv_config*. The setup function identified by the selected ppcv_chip is responsible for filling in the fields with the appropriate routines for the chip. The variable is statically initialized with a set of do-nothing routines, so if a particular chip doesn't need something done in one spot (typically the *cpuconfig[1/2]* routines), the setup routine doesn't have to fill anything in).

The general design rules for the routines are that they should perform whatever chip-specific actions that they can perform that are not also board-specific. For example, the old startup *main()* functions would sometimes turn off data translation, since some IPLs turned it on. With the new startups this is handled automatically by the library. On the other hand, both the old and new startups call the *ppc700_init_l2_cache()* manually in *main()*, since the exact bits to put in the L2CR register are board-specific. The routines in the libraries should be modified to work with the IPL and initialize the CPU properly, rather than modifying the board-specific code to hack around it (e.g. the aforementioned disabling of data translation).

The setup routine might also initialize a couple of other freestanding variables that other support routines use to avoid them having to check the PVR value again (e.g. see the *ppc600_set_clock_freqs()* and *ppcv_setup_7450()* functions for an example).

The new startup (and kernel, when used with a new startup) no longer depends on the PVR to identify the chip family. Instead the "family" field is filled in with a PPC_FAMILY_* value from **<ppc/syspage.h>**. This is transferred to the **ppc_kerinfo_entry.family** field on the system page, which the kernel uses to verify that the right version of **procnto** is being used.

If the kernel sees a value of PPC_FAMILY_UNKNOWN (zero) in the system page, it assumes that an old style startup is being used and will attempt to determine the family (and **cpuinfo->flags**) fields on its own. DO NOT USE that feature with new startups.

Fill in the *ppcv_config.family* and *ppcv_chip.cpu_flags* field properly. The *cpuconfig1* routine is used to configure the CPU for use in startup, and is called early before *main()* is called. For example, it makes sure that instruction and data translation is turned off, the exception table is pointed at low memory, etc. It's called once for every CPU in an SMP system, with the *cpu* parm indicating the CPU number being initialized.

The *cpuconfig2* routine is called just before startup transfers control to the first bootstrap executable in the image file system. It configures the CPU for running in the bootstrap environment, e.g. turning on CPU-specific features such as HID0 and HID1 bits. Again it's called once per CPU in an SMP system with the *cpu* parm indicating which one.

The *cpuinfo* routine is called by *init_one_cpuinfo()* to fill in the **cpuinfo_entry** structure for each CPU. The *qtime* routine is called by *init_qtime()* to set up the *qtime* syspage section.

The *map* and *unmap* routines used to create/delete memory mappings for startup and callout use, are called by:

- *startup_map_io*
- *startup_map_memory*
- *startup_unmap_io*
- *startup_unmap_memory*
- *callout_map_io*
- *callout_map_memory*

There's one more data variable to mention. This is *ppcv_list*, which is a statically initialized array of pointers to *ppcv_chip* structures. The default version of the variable in the library has a list of all the *ppcv_chip* variables defined by the library so, by default, the library is capable of handling any type of PPC chip.

By defining a *ppcv_list* variable in the board-specific directory and adding only the *ppcv_chip_** variable(s) that can be used with that board, all the chip-specific code for the processors that can't possibly be there will be left out.

For example, the new shasta-ssc startup with the default *ppcv_list* is about 1 KB bigger than the old version. By restricting the *ppcv_list* to only *ppcv_chip_750*, the new startup drops to 1 KB smaller than the original.

Adding a new CPU to the startup library

For a CPU called *xyz*, create a **<ppcv_chip_xyz.c>** and in it put an appropriately initialized struct *ppcv_chip ppcv_chip_xyz* variable. Add the *ppcv_chip_xyz* variable to the default **ppcv_list** (in **<ppcv_list.c>**).

If you were able to use an already existing *ppcv_setup_*()* function for the *ppcv_chip_xyz* initialization, you're done. Otherwise, create a **<ppcv_setup_xyz.c>** file with the properly coded *ppcv_setup_xyz()* function in it (don't forget to add the prototype to **<cpu_startup.h>**).

If you were able to use already existing **ppcv_*** routines in the *ppcv_setup_xyz()* function, you're done. Otherwise, create the routines in the appropriate **<ppcv_*_xyz.c>** files (don't forget to add the prototype(s) to **<cpu_startup.h>**). When possible, code the routines in an object-oriented manner, calling already existing routines to fill more generic information, e.g. *ppcv_cpuconfig2_700()* uses *ppcv_cpuconfig2_600()* to do most of the work and then it just fills in the 700 series-specific info.

With the new design, the following routines are now deprecated (and they spit out a message to that effect if you call them):

ppc600_init_features(), ppc600_init_caches(), ppc600_flush_caches()

Handled automatically by the library now.

ppc7450_init_l2_cache()

Use *ppc700_init_l2_cache()* instead.

Chapter 6

Customizing the Flash Filesystem

In this chapter. . .

Introduction

Neutrino ships with a small number of prebuilt flash filesystem drivers for particular embedded systems. For the currently available drivers, look in the **${QNX_TARGET}/${PROCESSOR}/sbin** directory. The flash filesystem drivers are named **devf-***system*, where *system* is derived from the name of the embedded system. You'll find a general description of the flash filesystem in the *System Architecture* book and descriptions of all the flash filesystem drivers in the *Utilities Reference*.

If a driver isn't provided for your particular target embedded system, you should first try our "generic" driver (**devf-generic**). This driver often — but not always — works with standard flash hardware. The driver assumes a supported memory technology driver (MTD) and linear memory addressing.

If none of our drivers works for your hardware, you'll need to build your own driver. We provide all the source code needed for you to customize a flash filesystem driver for your target. After installation, look in the *bsp_working_dir***/src/hardware/flash/boards** directory — you'll find a subdirectory for each board we support.

Besides the **boards** directory, you should also refer to the following sources to find out what boards/drivers we currently support:

- QNX docs (BSP docs as well as **devf-*** entries in *Utilities Reference*)
- the Community area of our website, **www.qnx.com**

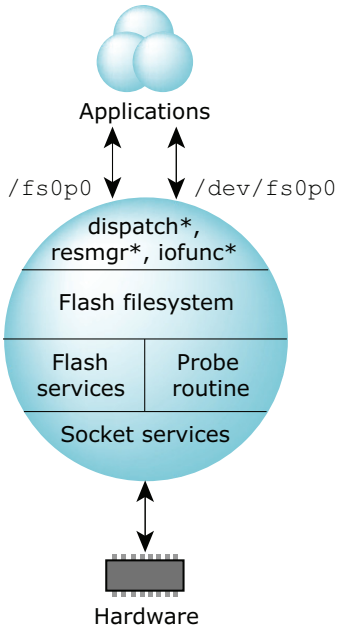
Note that we currently support customizing a driver only for embedded systems with onboard flash memory (also called a resident flash array or RFA). If you need support for *removable media* like PCMCIA or compact or miniature memory cards, then please contact us.

Driver structure

Every flash filesystem driver consists of the following components:

- *dispatch*, *resmgr*, and *iofunc* layers
- flash filesystem
- socket services
- flash services
- probe routine

When customizing the flash filesystem driver for your system, you'll be modifying the *main()* routine for the flash filesystem and providing an implementation of the socket services component. The other components are supplied as libraries to link into the driver.



Structure of the flash filesystem driver.

resmgr and *iofunc* layers

Like all Neutrino device managers, the flash filesystem uses the standard *resmgr*/*iofunc* interface and accepts the standard set of resource manager messages. The flash filesystem turns these messages into read, write, and erase operations on the underlying flash devices.

For example, an *open* message would result in code being executed that would read the necessary filesystem data structures on the flash device and locate the requested file. A subsequent *write* message will modify the contents of the file on flash. Special functions, such as erasing the flash device, are implemented using *devctl* messages.

Flash filesystem component

The flash filesystem itself is the "personality" component of the flash filesystem driver. The filesystem contains all the code to process filesystem requests and to manage the filesystem on the flash devices. The socket and flash services components are used by the flash filesystem to access the flash devices.

The code for the flash filesystem component is platform-independent and is provided in the **`libfs-flash3.a`** library.

Socket services component

The socket services component is responsible for any system-specific initialization required by the flash devices at startup and for providing addressability to the flash devices (this applies mainly to windowed flash interfaces).

Before reading/writing the flash device, other components will use socket services to make sure the required address range can be accessed. On systems where the flash device is linearly mapped into the processor address space, addressability is trivial. On systems where the flash is either bank-switched or hidden behind some other interface (such as PCMCIA), addressability is more complicated.

The socket services component is the one that will require the most customization for your system.

Flash services component

The flash services component contains the device-specific code required to write and erase particular flash devices. This component is also called the memory technology driver (MTD).

The directory **`${QNX_TARGET}/${PROCESSOR}/lib`** contains the MTD library **`libmtd-flash.a`** to handle the flash devices we support.

bsp_working_dir`/src/hardware/flash/mtd-flash` contains source for the **`libmtd-flash.a`** library.

Probe routine component

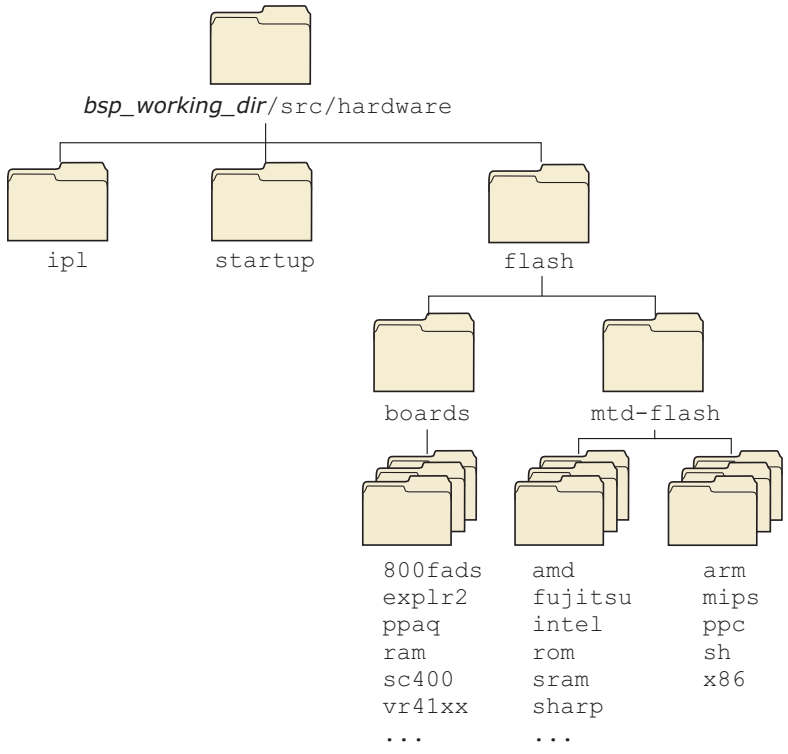
The probe routine uses a special algorithm to estimate the size of the flash array. Since the source code for the probe routine is available, you should be able to readily identify any failures in the sizing algorithm.

Building your flash filesystem driver

Before you start customizing your own flash filesystem driver, you should examine the source of all the sample drivers supplied. Most likely, one of the existing drivers can be easily customized to support your system. If not, the **`devf-ram`** source provides a good template to start with.

The source tree

The source files are organized as follows:



Flash directory structure.

The following pathnames apply to the flash filesystems:

Pathname	Description
`${QNX_TARGET}/usr/include/sys`	Header file **`f3s_mtd.h`**.
`${QNX_TARGET}/usr/include/fs`	Header files **`f3s_api.h`**, **`f3s_socket.h`**, and **`f3s_flash.h`**.
`${QNX_TARGET}/${PROCESSOR}/lib`	Libraries for flash filesystem and flash services.
*bsp_working_dir***`/src/hardware/flash/boards`**	Source code for socket services.
*bsp_working_dir***`/src/hardware/flash/mtd-flash`**	Source code for flash services as well as for probe routine and helper functions.

Before you modify any source, you should:

1. Create a new directory for your driver in the *bsp_working_dir***`/src/hardware/flash/boards`** directory.
2. Copy the files from the sample directory you want into your new directory.

For example, to create a driver called **`myboard`** based on the 800FADS board example, you would:

```
cd bsp_working_dir/hardware/flash/boards
mkdir myboard
cp -cRv 800fads myboard
cd myboard
make clean
```

The copy command (**cp**) specifies a recursive copy (the **-R** option). This will copy all files from the specified source directory *including* the subdirectory indicating which CPU this driver should be built for. In our example above, the **800fads** directory has a **ppc** subdirectory — this will cause the new driver (**myboard** in our example) to be built for the PowerPC.

The `Makefile`

When you go to build your new flash filesystem driver, you don't need to change the **Makefile**. Our recursive makefile structure ensures you're linking to the appropriate libraries.

Making the driver

You should use the following command to make the driver:

```
make F3S_VER=3 MTD_VER=2
```

For more information, see the technical note *Migrating to the New Flash Filesystem*.

The *main()* function

The *main()* function for the driver, which you'll find in the **main.c** file in the sample directories, is the first thing that needs to be modified for your system. Let's look at the **main.c** file for the 800FADS board example:

```
/*
** File: main.c for 800FADS board
*/
#include <sys/f3s_mtd.h>
#include "f3s_800fads.h"

int main(int argc, char **argv)
{
   int error;
   static f3s_service_t service[]=
   {
      {
         sizeof(f3s_service_t),
         f3s_800fads_open,
         f3s_800fads_page,
         f3s_800fads_status,
         f3s_800fads_close
      },
      {
         /* mandatory last entry */
         0, 0, 0, 0, 0
      }
   };
```

```
    static f3s_flash_v2_t flash[] =
    {
        {
            sizeof(f3s_flash_v2_t),
            f3s_a29f040_ident,      /* Common Ident             */
            f3s_a29f040_reset,      /* Common Reset             */

            /* v1 Read/Write/Erase/Suspend/Resume/Sync (Unused) */
            NULL, NULL, NULL, NULL, NULL, NULL,

            NULL,                   /* v2 Read (Use default)    */

            f3s_a29f040_v2write,    /* v2 Write                 */
            f3s_a29f040_v2erase,    /* v2 Erase                 */
            f3s_a29f040_v2suspend,  /* v2 Suspend               */
            f3s_a29f040_v2resume,   /* v2 Resume                */
            f3s_a29f040_v2sync,     /* v2 Sync                  */

            /* v2 Islock/Lock/Unlock/Unlockall (not supported)  */
            NULL, NULL, NULL, NULL
        },

        {
            /* mandatory last entry */
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

        }
    };

    /* init f3s */
    f3s_init(argc, argv, flash);

    /* start f3s */
    error = f3s_start(service, flash);

    return error;
}
```

The *service* array contains one or more **f3s_service_t** structures, depending on how many different sockets your driver has to support. The **f3s_service_t** structure, defined in **<fs/f3s_socket.h>**, contains function pointers to the socket services routines.

The *flash* array contains one or more **f3s_flash_t** structures, depending on how many different types of flash device your driver has to support. The **f3s_flash_t** structure, defined in **<fs/f3s_flash.h>**, contains function pointers to the flash services routines.

The *f3s_init()* and *f3s_start()* functions are defined in the **<fs/f3s_api.h>** header file.

Don't use the **<fs/f3s_socket.h>**, **<fs/f3s_flash.h>**, and **<fs/f3s_api.h>** header files directly. Instead, you should include **<sys/f3s_mtd.h>** for backward and forward compatibility.

f3s_init()

```
f3s_init (int argc,
          char **argv,
          f3s_flash_t *flash_vect)
```

This function passes the command-line arguments to the flash filesystem component, which then initializes itself.

f3s_start()

```
f3s_start (f3s_service_t *service,
           f3s_flash_t *flash)
```

This function passes the *service* and *flash* arrays to the filesystem component so it can make calls to the socket and flash services, and then starts the driver. This function returns only when the driver is about to exit.

When writing your **`main.c`**, you'll need to enter:

- the socket services functions for each socket in the *service* array
- the flash services functions for each flash device in the *flash* array.

If you have a system with only one socket consisting of the same flash devices, then there will be only a single entry in each array.

Socket services interface

The socket services interface, defined in the **`<fs/f3s_socket.h>`** header file, consists of the following functions:

- *f3s_open()*
- *f3s_page()*
- *f3s_status()*
- *f3s_close()*
- *f3s_socket_option()*
- *f3s_socket_syspage()*

f3s_open()

```
int32_t f3s_open (f3s_socket_t *socket,
                  uint32_t flags)
```

This function is called to initialize a socket or a particular window in a socket. The function should process any socket options, initialize and map in the flash devices, and initialize the *socket* structure.

f3s_page()

```
uint8_t *f3s_page (f3s_socket_t *socket,
                   uint32_t flags,
                   uint32_t offset,
                   int32_t *size)
```

This function is called to access a *window_size* sized window at address *offset* from the start of the device; it must be provided for both bank-switched and linearly mapped flash devices. If the *size* parameter is non-NULL, you should set it to the size of the window. The function must return a pointer suitable for accessing the device at address *offset*. On error, it should return NULL and set *errno* to ERANGE.

f3s_status()

```
int32_t f3s_status (f3s_socket_t *socket,
                    uint32_t flags)
```

This function is called to get the socket status. It's used currently only for interfaces that support dynamic insertion and removal. For onboard flash, you should simply return EOK.

f3s_close()

```
void f3s_close (f3s_socket_t *socket,
                uint32_t flags)
```

This function is called to close the socket. If you need to, you can disable the flash device and remove any programming voltage, etc.

The following flags are defined for the *flags* parameter in the socket functions:

F3S_POWER_VCC — Apply read power.

F3S_POWER_VPP — Apply program power.

F3S_OPER_SOCKET — Operation applies to socket given in *socket_index*.

F3S_OPER_WINDOW — Operation applies to window given in *window_index*.

The *socket* parameter is used for passing arguments and returning results from the socket services and for storing information about each socket. To handle complex interfaces such as PCMCIA, the structure has been defined so that there can be more than one socket; each socket can have more than one window. A simple linear flash array would have a single socket and no windows.

The *socket* structure is defined as:

```
typedef struct f3s_socket_s
{
  /*
   * these fields are initialized by the flash file system
   * and later validated and set by the socket services
   */
  _Uint16t struct_size;    /* size of this structure */
  _Uint16t status;         /* status of this structure */
  _Uint8t *option;         /* option string from flashio */
  _Uint16t socket_index;   /* index of socket */
  _Uint16t window_index;   /* index of window */

  /*
   * these fields are initialized by the socket services and later
```

```
   * referenced by the flash file system
   */
  _Uint8t *name;           /* name of driver */
  _Paddr64t address;       /* physical address 0 for allocated */
  _Uint32t window_size;    /* size of window power of two mandatory */
  _Uint32t array_offset;   /* offset of array 0 for based */
  _Uint32t array_size;     /* size of array 0 for window_size */
  _Uint32t unit_size;      /* size of unit 0 for probed */
  _Uint32t flags;          /* flags for capabilities */
  _Uint16t bus_width;      /* width of bus */
  _Uint16t window_num;     /* number of windows 0 for not windowed */

  /*
   * these fields are initialized by the socket services and later
   * referenced by the socket services
   */
  _Uint8t* memory;         /* access pointer for window memory */
  void *socket_handle;     /* socket handle pointer for external
                              library */
  void *window_handle;     /* window handle pointer for external
                              library */

  /*
   * this field is modified by the socket services as different window
   * pages are selected
   */
  _Uint32t window_offset;  /* offset of window */
}
f3s_socket_t;
```

Here's a description of the fields:

option	Option string from command line; parse using the *f3s_socket_option()* function.
socket_index	Current socket.
window_index	Current window.
name	String containing name of driver.
address	Base address of flash array.
window_size	Size of window in bytes.
array_size	Size of array in bytes; 0 indicates unknown.
unit_size	Size of unit in bytes; 0 indicates probed.
flags	The *flags* field is currently unused.
bus_width	Width of the flash devices in bytes.
window_num	Number of windows in socket; 0 indicates non-windowed.
memory	Free for use by socket services; usually stores current window address.

socket_handle	Free for use by socket services; usually stores pointer to any extra data for socket.
window_handle	Free for use by socket services; usually stores pointer to any extra data for window.
window_offset	Offset of window from base of device in bytes.

Options parsing

The socket services should parse any applicable options before initializing the flash devices in the *f3s_open()* function. Two support functions are provided for this:

f3s_socket_option()

```
int f3s_socket_option (f3s_socket_t *socket)
```

Parse the driver command-line options that apply to the socket services.

Currently the following options are defined:

```
-s baseaddress,windowsize, arrayoffset, arraysize, unitsize, buswidth, interleave
```

where:

baseaddress	Base address of the socket/window.
windowsize	Size of the socket/window.
arrayoffset	Offset of window from base of devices in bytes.
arraysize	Size of array in bytes, 0 indicates unknown.
buswidth	Memory bus attached to the flash chips.
interleave	Number of physical chips interleaved to form a larger logical chip (e.g. two 16-bit chips interleaved to form a 32-bit logical chip).

f3s_socket_syspage()

```
int f3s_socket_syspage (f3s_socket_t *socket)
```

Parse the syspage options that apply to the socket services.

The *syspage* options allow the socket services to get any information about the flash devices in the system that is collected by the startup program and stored in the syspage. See the chapter on Customizing Image Startup Programs for more information.

Flash services interface

The flash services interface, defined in the **<fs/f3s_flash.h>** header file, consists of the following functions:

- *f3s_ident()*
- *f3s_reset()*

- *f3s_v2read()*
- *f3s_v2write()*
- *f3s_v2erase()*
- *f3s_v2suspend()*
- *f3s_v2resume()*
- *f3s_v2sync()*
- *f3s_v2islock()*
- *f3s_v2lock()*
- *f3s_v2unlock()*
- *f3s_v2unlockall()*

The values for the *flags* parameter are defined in **<fs/s3s_flash.h>**. The most important one is F3S_VERIFY_WRITE. If this is set, the routine must perform a read-back verification after the write as a double check that the write succeeded. Occasionally, however, the hardware reports success even when the write didn't work as expected.

f3s_ident()

```
int32_t f3s_ident (f3s_dbase_t *dbase,
                   f3s_access_t *access,
                   uint32_t text_offset,
                   uint32_t flags)
```

Identifies the flash device at address *text_offset* and fills in the *dbase* structure with information about the device type and geometry.

f3s_reset()

```
void f3s_reset (f3s_dbase_t *dbase,
                f3s_access_t *access,
                uint32_t text_offset)
```

Resets the flash device at address *text_offset* into the default read-mode after calling the *fs3_ident()* function or after a device error.

f3s_v2read()

```
int32_t f3s_v2read (f3s_dbase_t *dbase,
                    f3s_access_t *access,
                    _Uint32t flags,
                    _Uint32t text_offset,
                    _Int32t buffer_size,
                    _Uint8t *buffer);
```

This optional function is called to read *buffer_size* bytes from address *text_offset* into *buffer*. Normally the flash devices will be read directly via *memcpy()*.

On success, it should return the number of bytes read. If an error occurs, it should return -1 with *errno* set to one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power, but corruption is localized and block will be usable after erasing).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2write()

```
int32_t f3s_v2write (f3s_dbase_t *dbase,
                     f3s_access_t *access,
                     _Uint32t flags,
                     _Uint32t text_offset,
                     _Int32t buffer_size,
                     _Uint8t *buffer);
```

This function writes *buffer_size* bytes from *buffer* to address *text_offset*.

On success, it should return the number of bytes written. If an error occurs, it should return -1 with *errno* set to one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or write failed, but corruption is localized and block will be usable after erasing).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EROFS	Block is write protected.
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2erase()

```
int f3s_v2erase (f3s_dbase_t *dbase,
                 f3s_access_t *access,
                 _Uint32t flags,
                 _Uint32t text_offset);
```

This function begins erasing the flash block containing the *text_offset*. It can optionally determine if an error has already occurred, or it can just return EOK and let *f3s_v2sync()* detect any error.

On success, it should return EOK. If an error occurs, it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or erase failed, but corruption is localized and block will be usable after an erase)
EFAULT	Unrecoverable I/O error (e.g. block no longer usable)
EROFS	Block is write protected
EINVAL	Invalid command error
EBUSY	Flash busy, try again (e.g. erasing same block twice)
ERANGE	Flash memory access out of range (via service->page function)
ENODEV	Flash no longer accessible (e.g. flash removed)
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2suspend()

```
int f3s_v2suspend (f3s_dbase_t *dbase,
                   f3s_access_t *access,
                   _Uint32t flags,
                   _Uint32t text_offset);
```

This function suspends an erase operation, when supported, for a read or for a write.

On success, it should return EOK. If an error occurs, it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or erase failed, but corruption is localized and block will be usable after erasing).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ECANCELED	Suspend canceled because erase has already completed.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2resume()

```
int f3s_v2resume (f3s_dbase_t *dbase,
                  f3s_access_t *access,
                  _Uint32t flags,
                  _Uint32t text_offset);
```

This function resumes an erase operation after a suspend command has been issued.

On success, it should return EOK. If an error occurs, it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or erase failed, but corruption is localized and block will be usable after erasing).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2sync()

```
int f3s_v2sync (f3s_dbase_t *dbase,
                f3s_access_t *access,
                _Uint32t flags,
                _Uint32t text_offset);
```

This function determines whether an erase operation has completed and returns any detected error.

On success, it should return EOK. If an error occurs, it should return one of the following:

EAGAIN	Still erasing.
EIO	Recoverable I/O error (e.g. failed due to low power or erase failed, but corruption is localized and block will be usable after an erase).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EROFS	Block is write protected.
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2islock()

```
int f3s_v2islock (f3s_dbase_t *dbase,
                  f3s_access_t *access,
                  _Uint32t flags,
                  _Uint32t text_offset);
```

This function determines whether the block containing the address *text_offset* can be written to (we term it as success) or not.

On success, it should return EOK. If the block cannot be written to, it should return EROFS. Otherwise, an error has occurred and it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or lock failed, but corruption is localized and block will be usable after an erase).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2lock()

```
int f3s_v2lock (f3s_dbase_t *dbase,
                f3s_access_t *access,
                _Uint32t flags,
                _Uint32t text_offset);
```

This function write-protects the block containing the address *text_offset* (if supported). If the block is already locked, it does nothing.

On success, it should return EOK. If an error occurs, it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or lock failed, but corruption is localized and block will be usable after an erase).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2unlock()

```
int f3s_v2unlock (f3s_dbase_t *dbase,
                  f3s_access_t *access,
                  _Uint32t flags,
                  _Uint32t text_offset);
```

This function clears write-protection of the block containing the address *text_offset* (if supported). If the block is already unlocked, it does nothing. Note that some devices do not support unlocking of arbitrary blocks. Instead all blocks must be unlocked at the same time. In this case, use *f3s_v2unlockall()* instead.

On success, it should return EOK. If an error occurs, it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or unlock failed, but corruption is localized and block will be usable after an erase).

EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

f3s_v2unlockall()

```
int f3s_v2unlockall (f3s_dbase_t *dbase,
                     f3s_access_t *access,
                     _Uint32t flags,
                     _Uint32t text_offset);
```

This function clears all write-protected blocks on the device containing the address *text_offset*. Some boards use multiple chips to form one single logical device. In this situation, each chip will have *f3s_v2unlockall()* invoked on it separately.

On success, it should return EOK. If an error occurs, it should return one of the following:

EIO	Recoverable I/O error (e.g. failed due to low power or unlock failed, but corruption is localized and block will be usable after an erase).
EFAULT	Unrecoverable I/O error (e.g. block no longer usable).
EINVAL	Invalid command error.
ERANGE	Flash memory access out of range (via service->page function).
ENODEV	Flash no longer accessible (e.g. flash removed).
ESHUTDOWN	Critical error; shut down the flash driver.

We currently don't support user-customized flash services, nor do we supply detailed descriptions of the flash services implementation.

Choosing the right routines

We provide several device-specific variants of the core set of flash services:

- *f3s_ident()*
- *f3s_reset()*
- *f3s_v2write()*
- *f3s_v2erase()*

- *f3s_v2suspend()*
- *f3s_v2resume()*
- *f3s_v2sync()*
- *f3s_v2islock()*
- *f3s_v2lock()*
- *f3s_v2unlock()*
- *f3s_v2unlockall()*.

For example, if you have a 16-bit Intel device and you want to use *f3s_v2erase()*, you'd use the *f3s_iCFI_v2erase()* routine.

For more information, see the technical note *Choosing the correct MTD Routine for the Flash Filesystem*.

The file **<sys/f3s_mtd.h>** can be found in:

*bsp_working_dir***/src/hardware/flash/mtd-flash/public/sys/f3s_mtd.h.**

Example: The `devf-ram` driver

This driver uses main memory rather than flash for storing the flash filesystem. Therefore, the filesystem is *not* persistent — all data is lost when the system reboots or **/dev/shmem/fs0** is removed. This driver is used mainly for test purposes.

main()

In the *main()* function, we declare a single *services* array entry for the socket services functions and a null entry for the flash services functions.

```
/*
** File: f3s_ram_main.c
**
** Description:
**
** This file contains the main function for the f3s
** flash filesystem
**
*/
#include "f3s_ram.h"

int main(int argc, char **argv)
{
   int error;
   static f3s_service_t service[] =
   {
      {
         sizeof(f3s_service_t),
         f3s_ram_open,
         f3s_ram_page,
```

```
            f3s_ram_status,
            f3s_ram_close
        },

        {
            /* mandatory last entry */
            0, 0, 0, 0, 0
        }
    };

    static f3s_flash_v2_t flash[] =
    {
        {
            sizeof(f3s_flash_v2_t),
            f3s_sram_ident,          /* Common Ident            */
            f3s_sram_reset,          /* Common Reset            */
            NULL,                    /* v1 Read (Deprecated)    */
            NULL,                    /* v1 Write (Deprecated)   */
            NULL,                    /* v1 Erase (Deprecated)   */
            NULL,                    /* v1 Suspend (Deprecated) */
            NULL,                    /* v1 Resume  (Deprecated) */
            NULL,                    /* v1 Sync (Deprecated)    */
            NULL,                    /* v2 Read (Use default)   */
            f3s_sram_v2write,        /* v2 Write                */
            f3s_sram_v2erase,        /* v2 Erase                */
            NULL,                    /* v2 Suspend (Unused)     */
            NULL,                    /* v2 Resume (Unused)      */
            f3s_sram_v2sync,         /* v2 Sync                 */
            f3s_sram_v2islock,       /* v2 Islock               */
            f3s_sram_v2lock,         /* v2 Lock                 */
            f3s_sram_v2unlock,       /* v2 Unlock               */
            f3s_sram_v2unlockall     /* v2 Unlockall            */
        },

        {
            /* mandatory last entry */
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        }
    };

    /* init f3s */
    f3s_init(argc, argv, (f3s_flash_t *)flash);

    /* start f3s */
    error = f3s_start(service, (f3s_flash_t *)flash);

    return (error);
}
```

f3s_ram_open()

In the socket services *open()* function, we assign a name for the driver and then process any options. If no options are specified, a default size is assigned and the memory for the (virtual) flash is allocated.

```
/*
** File: f3s_ram_open.c
**
** Description:
**
```

```
** This file contains the open function for the ram library
**
*/
#include "f3s_ram.h"

int32_t f3s_ram_open(f3s_socket_t *socket,
                     uint32_t flags)
{
   static void *   memory;
   char           name[8];
   int                 fd;
   int               flag;

   /* check if not initialized */
   if (!memory)
   {
      /* get io privileges */
      ThreadCtl(_NTO_TCTL_IO, NULL);

      /* setup socket name */
      socket->name = "RAM (flash simulation)";

      /* check if there are socket options */
      if (f3s_socket_option(socket))
         socket->window_size = 1024 * 1024;

      /* check if array size was not chosen */
      if (!socket->array_size)
         socket->array_size = socket->window_size;

      /* check if array size was not specified */
      if (!socket->array_size) return (ENXIO);

      /* set shared memory name */
      sprintf(name, "/fs%X", socket->socket_index);

      /* open shared memory */
      fd = shm_open(name, O_CREAT | O_RDWR, 0777);

      if (fd < 0) return (errno);

      /* set size of shared memory */
      flag = ftruncate(fd, socket->array_size);

      if (flag)
      {
         close(fd);
         return (errno);
      }

      /* map physical address into memory */
      memory = mmap(NULL, socket->array_size,
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, socket->address);

      if (!memory)
      {
         close(fd);
         return (errno);
      }

      /* copy socket handle */
      socket->socket_handle = (void *)fd;
```

```
    }

    /* set socket memory pointer to previously initialized
       value */
    socket->memory = memory;
    return (EOK);
}
```

f3s_ram_page()

In the socket services *page()* function, we first check that the given *offset* doesn't exceed the bounds of the allocated memory, and then assign the window *size* if required. The function returns the offset address modulo the window size.

```
/*
** File: f3s_ram_page.c
**
** Description:
**
** This file contains the page function for the ram library
**
*/
#include "f3s_ram.h"

uint8_t *f3s_ram_page(f3s_socket_t *socket,
                      uint32_t flags,
                      uint32_t offset,
                      int32_t *size)
{
    /* check if offset does not fit in array */
    if (offset >= socket->window_size)
    {
        errno = ERANGE;
        return (NULL);
    }

    /* select proper page */
    socket->window_offset = offset & ~(socket->window_size - 1);

    /* set size properly */
    *size = min((offset & ~(socket->window_size - 1)) +
                socket->window_size - offset, *size);

    /* return memory pointer */
    return (socket->memory + offset);
}
```

The socket services *status()* and *close()* don't do anything interesting in this driver.

Appendix A

System Design Considerations

In this appendix. . .

Introduction

Since Neutrino is a protected-mode 32-bit operating system, many limiting design considerations won't apply (particularly on the x86 platform, which is steeped in DOS and 8088 legacy concerns). By noting the various "do's" and "don'ts" given in this appendix, you'll be able to design and build an embedded system tailored for Neutrino.

You may also be able to realize certain savings, in terms of design time, hardware costs, and software customization effort.

Before you design your system

Before you begin designing your system, here are some typical questions you might consider:

- What speed of processor do you need?
- How much memory is required?
- What peripherals are required?
- How will you debug the platform?
- How will you perform field upgrades?

Naturally, your particular system will dictate whether all of these (or others) are relevant. But for the purposes of this discussion, we'll assume all these considerations apply.

Processor speed

Although Neutrino is a realtime operating system, this fact alone doesn't necessarily mean that any given *application* will run quickly. Graphical user interface applications can consume a reasonable amount of CPU and are particularly sensitive to the end-user's perception of speed.

If at all possible, try to prototype the system on either a standard PC (in the case of x86-based designs) or a supported evaluation board (in the case of x86, PPC, ARM, SH, and MIPS designs). This will very quickly give you a "feel" for the speed of a particular processor.

Memory requirements

During initial prototyping, you should plan on more memory on the target than during the final stages. This is because you'll often be running *debugging versions* of software, which may be larger. Also, you'll want to include diagnostics and utility programs, which again will consume more memory than expected. Once your prototype system is up and running, you can then start thinking about how much memory you "really" need.

Peripherals

Given a choice, you should use peripherals that are listed as supported by Neutrino. This includes such items as disk controllers, network cards, PC-Card controllers, flash memory chips, and graphics controllers. For lists of supported hardware, see the Community area of our website, **`http:www.qnx.com`**; for information about third-party products, see the Download area.

Graphics controllers are one of the particularly delicate areas in the design of an embedded system, often because a chip may be very new when it's selected and we may not yet have a driver for it. Also, if you're using a graphics controller in conjunction with an LCD panel, beware that this is perhaps the most complicated setup because of the many registers that must be programmed to make it work.

Note that QNX Software Systems can do custom development work for you; for more information, contact your sales representative. Other consulting houses offer similar services to the QNX community.

Debugging

In many cases, especially in cost-sensitive designs, you won't want to provide any additional functionality beyond that absolutely required for the project at hand. But since the project is usually a brand new design, you'll need to ensure that the hardware actually works *per se* and then actually works with the software.

We recommend that you install some form of easy-to-get-at hardware debugging port, so that the software can output diagnostics as it's booting. Generally, something as simple as a latched output that can drive a single LED is sufficient, but an 8- or 16-bit port that drives a number of 7-segment LEDs would be even better. Best of all is a simple serial port, because more meaningful diagnostics can be written by the software and easily captured.

This debug port can be left off for final assembly or a slightly modified "final" version of the board can be created. The cost savings in terms of software development time generally pay for the hardware modifications many times over.

Field upgrades

You can handle the issue of field upgrades in various ways, depending on the nature of your particular target system:

- a JTAG port
- socketed Flash/EPROM devices
- a communications port.

You may need such a vehicle for your update software even during your initial software development effort. At this early phase, you'll effectively be performing "field upgrades" as your software is being developed.

Other design considerations

There are other design considerations that relate to both the hardware and software development process. In this section, we'll discuss some of the more common ones.

EPROM/Flash filesystem considerations

Solid-state mass storage can be located anywhere in the address space — it should be linearly mapped. In legacy designs (particularly x86), the mass storage device was often forced into a window of some size (typically from 8 KB to 64 KB), with additional hardware being required to map that window into the processor's address space. Additionally, this window was traditionally located in the first 1 MB of memory.

With a modern, 32-bit processor, the physical address space of the processor is usually sufficient to address the entire mass storage device. In fact, this makes the software easier by not having to worry about how to address the window-mapping hardware.

The two driving factors to be considered in the hardware design of solid-state media are cost and compatibility. If the medium is to be soldered onto the board, then there's little chance that it may need to be compatible with other operating systems. Therefore, simply map the entire medium into the address space of the processor and don't add additional hardware to perform windowing or bank switching.

Adhering to standards (e.g. PCMCIA, FFS2, etc.) for solid-state memory is also unnecessary — our Flash filesystem drivers know how to address and use just a raw Flash device.

When the time comes to decide on the logical layout of the flash memory chips, the tradeoff will be between the size of the erase block and the speed of access. By taking four flash devices and organizing them into a 32-bit wide bus, you gain speed. However, you also increase the erase block size by a factor of four (e.g. 256 KB erase blocks).

Note that we don't recommend trying to XIP out of flash memory that's being used for a flash filesystem. This is because the flash filesystem may need to erase a particular block of memory. While this erase operation is in progress, depending on the particular type of flash memory device you have, the *entire* device may be unusable. If this is *also* the device containing the code that the processor is actively executing from, you'll run into problems. Therefore, we recommend that you use at least two independent sets of flash devices: one set for the filesystem and one set for the code.

IPL location

Under Neutrino, the only location requirement is that the ROM boot device that performs the IPL be addressable at the *processor's reset vector*. No special hardware is required to be able to "move" the location of the boot ROM.

Graphics cards

All the drivers under Neutrino can be programmed to deal with graphics hardware at any address — there's no requirement to map the VGA video aperture below 1 MB.

A20 gate

On the x86 platform, another vestige of the legacy 1 MB address limitation is usually found in something called an *A20 gate*. This is a piece of hardware that would force the A20 address line to zero, regardless of the actual setting of the A20 address line on the processor.

The justification for this was for legacy software that would depend on the ability to wrap past location **0xFFFFF** back to **0x00000**. Neutrino doesn't have such a requirement. As a result, the OS doesn't need any A20 gate hardware to be installed. Note that some embedded x86 processors have the A20 gate hardware built right into the processor chip itself — the IPL will disable the A20 gate as soon as possible after startup.

If your system requires a standard BIOS, there's a small chance that the BIOS will make use of the A20 gate. To find out for certain, consult your BIOS supplier.

External ISA bus slots

Neutrino doesn't require the external ISA bus to be mapped into the usual x86 **0x00000**-to-**0xFFFFF** address range. This simplifies the hardware design, eliminating issues such as shadow RAM and the requirement to move a portion of the RAM (usually **0xA0000** through **0xFFFFF**) to some other location.

But if your hardware needs to run with a standard BIOS and to support BIOS extensions, then this optimization can't be implemented, because the BIOS expects extensions at **0xA0000** through **0xEFFFF** (typically).

PCI bus slots

In Neutrino, all PCI drivers interface to a PCI resource manager (e.g. **pci-bios**, **pci-p5064**, **pci-raven**), which handles the hardware on behalf of the drivers.

For details, see the **pci-*** entries in the *Utilities Reference*.

External clocks

Neutrino can be driven with an external clock. In some systems there's a "standard" clock source supplied as part of the system or of the highly integrated CPU chip itself. For convenience, the OS can operate with an external clock source that's not generated by this component. However, keep two things in mind:

- The timing resolution for software timers will be no better than the timing resolution of the external clock.
- The hardware clock will be driving a software interrupt handler.

Therefore, keep the rates down to a reasonable number. Almost all modern processors can handle clock interrupts at 1 kHz or lower — processors with higher CPU clock rates (e.g. Pentium-class, 300 MHz RISC processors, etc.) can handle faster clock interrupts.

Note that there's no requirement to keep the clock frequency to some "round number." If it's convenient to derive the clock interrupt from a baud rate generator or other crystal, the OS will be able to accurately scale the incoming clock rate for use in its internal timers and time-of-day clocks.

Interrupts & controllers

On an x86 design, the default startup supports two Programmable Interrupt Controllers (PICs). These must be 8259-compatible, with the standard configuration of a secondary 8259 connected to the IRQ2 line of the primary interrupt controller.

Beware of hanging devices off IRQ7 and IRQ15 on an 8259 chip — these are generally known as the "glitch interrupts" and can be unreliable.

If your x86 hardware design differs, there's no constraint about the PICs, but you must write the code to handle them.

On non-x86 designs, be aware that there may be only one interrupt line going to the processor and that a number of hardware devices may be sharing that one line. This is generally accomplished in one of two ways:

- wire-OR
- PIC chip

In either case, the relevant design issue is to determine the ordering and priority of interrupts from hardware sources. You'll want to arrange the hardware and software to give highest priority (and first order) to the interrupt source that has the most stringent latency requirements. (For more details, see the chapter on Writing an Interrupt Handler in the *Programmer's Guide*, along with the *InterruptAttach()* and *InterruptAttachEvent()* function calls in the *Library Reference*.)

Serial and parallel ports

Serial and parallel ports are certainly desirable — and highly recommended — but not required. The 16550 component with 16-byte FIFOs is suitable for Neutrino. Our drivers can work with these devices on a byte-aligned or doubleword-aligned manner.

If you're going to support multiple serial ports on your device, you can have the multiple devices share the same interrupt. It's up to the software to decide which device generated the interrupt and then to handle that interrupt. The standard Neutrino serial port handlers are able to do this.

Although the serial driver can be told to use a "nonstandard" clock rate when calculating its divisor values, this can cause the baud rate to deviate from the standard.

Try to run DTR, DSR, RTS, CTS if possible, because hardware flow control will help on slower CPUs.

Parallel port considerations

Generally, the parallel port does *not* require an interrupt line — this isn't used by our standard parallel port drivers.

NMI

Avoid the *Non-Maskable Interrupt* (NMI) in x86 designs. PPC, MIPS, ARM, and SH-4 don't even support it.

An NMI is an interrupt which can't be disabled by clearing the CPU's interrupt enable flag, unlike most normal interrupts. Non-Maskable interrupts are typically used to signal events that require immediate action, such as a parity error, a hardware failure, or imminent loss of power.

The problem with NMIs is that they can occur even when interrupts have been disabled. This is important because sometimes it's assumed that interrupts can be masked to avoid being interrupted. NMIs undermine this assumption and this can lead to unexpected behaviour if an NMI fires during a period in which that software expects to be operating without interruption.

For this reason, NMIs are normally only used when the subsequent condition of the machine is not a relevant consideration; for instance, when the machine is about to shut down, or when an unrecoverable hardware error has occurred.

Anytime an NMI is used, any software may experience unexpected behavior and there's no good way to predict what the behavior may be.

Design do's and don'ts

Before you commit to a design, take a look at the following tips — you may save yourself some grief. Although some of these points assume you're relying on our Custom Engineering services, the principles behind all of them are sound.

Do:

- Do design in more speed/memory than you think you need.
- Do try a proof of concept using off-the-shelf hardware, if possible.
- Do have a serial port/debug output device on the board; have it reasonably close to the CPU in hardware terms (i.e. don't put it on the other side of a PCI bridge).
- Do allow the ROM/flash devices holding the IPL code to be socketed.
- If you're using a MIPS processor, make sure any devices needed by the IPL/startup sequence are in the physical address range of **`0x00000000`** to **`0x20000000`** — that makes it accessible from the **`kseg1`** virtual address block.

- Do consider staggering a device's ports by any power of 2, but don't mix up the address lines so that the I/O registers appear in a strange order.
- Do try to use a timer chip that allows free-running operation, rather than one that requires poking after every interrupt.
- Do put the timer on its own interrupt line so that the kernel doesn't have to check that the interrupt actually came from the timer.
- Do follow the CPU's interface for reporting a bus error — don't report it as a hardware interrupt.
- If you have optional pieces, make sure you have some positive method of determining what pieces are present (something other than poking at it and seeing if it responds).
- Do run your design by us, ideally *before* you build it.
- Do make a point of stating requirements you think are obvious.
- Do remember to point out any pitfalls you know about.
- Do send us as much documentation as you have available on chipsets, panels, etc.

Don't:

- Don't use write-only registers.
- Don't nest interrupt controller chips too deeply — one big wide interrupt controller is better.
- Don't use hardware that requires short delays between register accesses (e.g. Zilog SCC).
- Don't put information from many different places into the same I/O register location if the OS/drivers also have to do RMW cycles to it.
- Don't decide that no-BIOS is the way to go just because it sounds cool.
- Don't use a $2.00 chip instead of a $3.00 chip and expect the performance of a $10.00 chip.
- Don't build your first run of boards without leaving a way to debug the system.
- Don't build your first run of boards with only 1 MB of RAM on board.
- Don't send us anything without correct schematics that match what you send.
- Don't program the flash and then solder it on, leaving us with no option to reprogram it.
- Don't build just one prototype that must be shipped back and forth several times.

Appendix B

Sample Buildfiles

In this appendix. . .

Introduction

In this appendix, we'll look at some typical buildfiles you can use with **mkifs** or import into the IDE's System Builder to get your system up and running. This appendix is divided into two main parts:

- a "generic" part that contains some incomplete cut-and-paste fragments illustrating common techniques, as well as complete samples for the x86 platform.
- processor-specific notes.

We finish with a section for each of the supported processor platforms, showing you differences from the x86 samples and noting things to look out for.

Note that you should read both the section for your particular processor as well as the section on generic samples, because things like shared objects (which are required by just about everything) are documented in the generic section.

Generic examples

In this section, we'll look at some common buildfile examples that are applicable (perhaps with slight modifications, which we'll note) to all platforms. We'll start with some fragments that illustrate various techniques, and then we'll wrap up with a few complete buildfiles. In the "Processor-specific notes" section, we'll look at what needs to be different for the various processor families.

Shared libraries

The first thing you'll need to do is to ensure that the shared objects required by the various drivers you'll be running are present. *All* drivers require at least the standard C library shared object (**libc.so**). Since the shared object search order looks in **/proc/boot**, you don't have to do anything special, except include the shared library into the image. This is done by simply specifying the name of the shared library on a line by itself, meaning "include this file."

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

The following buildfile snippet applies:

```
# include the C shared library
libc.so
# create a symlink called ldqnx.so.2 to it
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
```

How do you determine which shared objects you need in the image? You can use the **objdump** utility to display information about the executables you're including in the image; look for the objects marked as **NEEDED** For example, suppose you're including **ping** in your image:

```
$ objdump -x 'which ping' | grep NEEDED
objdump: /usr/bin/ping: no symbols
  NEEDED      libsocket.so.2
  NEEDED      libc.so.3
```

The **ping** executable needs **libsocket.so.2** and **libc.so.3**. You need to use **objdump** recursively to see what these shared objects need:

```
$ objdump -x /lib/libsocket.so.2 | grep NEEDED
  NEEDED      libc.so.3
$ objdump -x /lib/libc.so.3 | grep NEEDED
```

The **libsocket.so.2** shared object needs only **libc.so.3**, which, in turn, needs nothing. So, if you're including **ping** in your image, you also need to include these two shared objects.

Running executables more than once

If you want to be able to run executables more than once, you'll need to specify the **[data=copy]** attribute for those executables. If you want it to apply to *all* executables, just put it on a line by itself before the executables. This causes the data segment to be copied before it's used, preventing it from being overwritten by the first invocation of the program.

Multiple consoles

For systems that have multiple consoles or multiple serial ports, you may wish to have the shell running on each of them. Here's an example showing you how that's done:

```
[+script] .script = {
    # start any other drivers you need here
    devc-con -e -n4 &
    reopen /dev/con1
    [+session] esh &
    reopen /dev/con2
    [+session] esh &
    ...
```

As you can see, the trick is to:

1. Start the console driver with the **-n** option to ask for more than one console (in this case, we asked for four virtual consoles).
2. Redirect standard input, output, and error to each of the consoles in turn.
3. Start the shell on each console.

It's important to run the shell in the background (via the ampersand character "&") — if you don't, then the interpretation of the script will suspend *until the shell exits!*

Starting other programs on consoles

Generally speaking, this method can be used to start various other programs on the consoles (that is to say, you don't have to start the shell; it could be *any* program).

To do this for serial ports, start the appropriate serial driver (e.g. **devc-ser8250**), and redirect standard input, output, and error for each port (e.g. **/dev/ser1**, **/dev/ser2**). Then run the appropriate executable (in the background!) after the redirection.

The **[+session]** directive makes the program the session leader (as per POSIX) — this may not be necessary for arbitrary executables.

Redirection

You can do the **reopen** on any device as many times as you want. You would do this, for example, to start a program on **/dev/con1**, then start the shell on **/dev/con2**, and then start another program on **/dev/con1** again:

```
[+script] .script = {
    ...
    reopen /dev/con1
    prog1 &
    reopen /dev/con2
    [+session] esh &
    reopen /dev/con1
    prog2 &
    ...
```

/tmp

To create the **/tmp** directory on a RAM-disk, you can use the following in your buildfile:

```
[type=link] /tmp = /dev/shmem
```

This will establish **/tmp** as a symbolic link in the process manager's pathname table to the **/dev/shmem** directory. Since the **/dev/shmem** directory is really the place where shared memory objects are stored, this effectively lets you create files on a RAM-disk — files created are, in reality, shared memory objects living in RAM.

Note that the line containing the link attribute (the **[type=link]** line) should be placed *outside* of the script file or boot file — after all, you're telling **mkifs** that it should create a file that just happens to be a link rather than a "real" file.

Complete example — minimal configuration

This configuration file does the bare minimum necessary to give you a shell prompt on the first serial port:

```
[virtual=ppcbe,srec] .bootstrap = {
    startup-rpx-lite -Dsmc1.115200.64000000.16
    PATH=/proc/boot procnto-800
}
[+script] .script = {
    devc-serppc800 -e -F -c64000000 -b115200 smc1 &
    reopen

    [+session] PATH=/proc/boot esh &
}

[type=link] /dev/console=/dev/ser1
```

```
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so

libc.so

[data=copy]
devc-serppc800
esh
# specify executables that you want to be able
# to run from the shell:  echo, ls, pidin, etc...
echo
ls
pidin
cat
cp
```

Complete example — flash filesystem

Let's now examine a complete buildfile that starts up the flash filesystem:

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

[+script] .script = {
    devc-con -e -n5 &
    reopen /dev/con1
    devf-i365sl -r -b3 -m2 -u2 -t4 &
    waitfor /fs0p0
    [+session] TERM=qansi PATH=/proc/boot:/bin esh &
}

[type=link] /tmp=/dev/shmem
[type=link] /bin=/fs0p0/bin
[type=link] /etc=/fs0p0/etc

libc.so
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
libsocket.so

[data=copy]

devf-i365sl
devc-con
esh
```

The buildfile's **.bootstrap** specifies the usual **startup-bios** and **procnto** (the startup program and the kernel). Notice how we set the **PATH** environment variable to point not only to **/proc/boot**, but also to **/bin** — the **/bin** directory is a link (created with the **[type=link]**) to the flash filesystem's **/fs0p0/bin** path.

In the **.script** file, we started up the console driver with five consoles, reopened standard input, output, and error for **/dev/con1**, and started the flash filesystem driver **devf-i365sl**. Let's look at the command-line options we gave it:

-r Enable fault recovery for dirty extents, dangling extents, and partial reclaims.

-b3 Enable background reclaim at priority **3**.

-u2 Specify the highest update level (**2**) to update files and directories.

-t4 Specify the highest number of threads. Extra threads will increase performance when background reclaim is enabled (with the **-b** option) and when multiple chips and/or spare blocks are available.

The **devf-i365sl** will automatically mount the flash partition as **/fs0p0**. Notice the process manager symbolic links we created at the bottom of the buildfile:

```
[type=link] /bin=/fs0p0/bin
[type=link] /etc=/fs0p0/etc
```

These give us **/bin** and **/etc** from the flash filesystem.

Complete example — disk filesystem

In this example, we'll look at a filesystem for rotating media. Notice the shared libraries that need to be present:

```
[virtual=x86,bios +compress] .bootstrap = {
    startup-bios
    PATH=/proc/boot:/bin LD_LIBRARY_PATH=/proc/boot:/lib:/dll procnto
}


[+script] .script = {
    pci-bios &
    devc-con &
    reopen /dev/con1
# Disk drivers
    devb-eide blk cache=2m,automount=hd0t79:/,automount=cd0:/cd &

# Wait for a bin for the rest of the commands
    waitfor /x86 10

# Some common servers
    pipe &
    mqueue &
    devc-pty &

# Start the main shell
    [+session] esh &
}

# make /tmp point to the shared memory area
[type=link] /tmp=/dev/shmem

# Redirect console messages
# [type=link] /dev/console=/dev/ser1

# Programs require the runtime linker (ldqnx.so) to be at
# a fixed location
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so

# Add for HD support
[type=link] /usr/lib/libcam.so.2=/proc/boot/libcam.so

# add symbolic links for bin, dll, and lib
# (files in /x86 with devb-eide)
```

```
[type=link] /bin=/x86/bin
[type=link] /dll=/x86/lib/dll
[type=link] /lib=/x86/lib

# We use the C shared lib (which also contains the runtime linker)
libc.so

# Just in case someone needs floating point and our CPU doesn't
# have a floating point unit
fpemu.so.2

# Include the hard disk shared objects so we can access the disk
libcam.so
io-blk.so

# For the QNX 4 filesystem
cam-disk.so
fs-qnx4.so

# For the UDF filesystem and the PCI
cam-cdrom.so
fs-udf.so
pci-bios

# Copy code and data for all executables after this line
[data=copy]

# Include a console driver, shell, etc.
esh
devb-eide
devc-con
```

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

In this buildfile, we see the startup command line for the **devb-eide** command:

```
devb-eide blk cache=2m,automount=hd0t79:/automount=cd0:/cd &
```

This line indicates that the **devb-eide** driver should start and then pass the string beginning with the **cache=** through to the end (except for the ampersand) to the block I/O file (**io-blk.so**). This will examine the passed command line and then start up with a 2-megabyte cache (the **cache=2m** part), automatically mount the partition identified by **hd0t79** (the first QNX filesystem partition) as the pathname **/hd**, and automatically mount the CD-ROM as **/cd**.

Once this driver is started, we then need to wait for it to get access to the disk and perform the mount operations. This line does that:

```
waitfor /ppcbe/bin
```

This waits for the pathname **/ppcbe/bin** to show up in the pathname space. (We're assuming a formatted hard disk that contains a valid QNX filesystem with **${QNX_TARGET}** copied to the root.)

Now that we have a complete filesystem with all the shipped executables installed, we run a few common executables, like the Pipe server.

Finally, the list of shared objects contains the **.so** files required for the drivers and the filesystem.

Complete example — TCP/IP with network filesystem

Here's an example of a buildfile that starts up an Ethernet driver, the TCP/IP stack, and the network filesystem:

```
[virtual=armle,elf +compress] .bootstrap = {
  startup-abc123 -vvv
  PATH=/proc/boot procnto
}
[+script] .script = {
    devc-ser8250 -e -b9600 0x1d0003f8,0x23 &
    reopen

#  Start the PCI server
   pci-abc123 &
   waitfor /dev/pci

#  Network drivers and filesystems
    io-pkt-v4 -dtulip-abc123 &
    waitfor /dev/socket
    ifconfig en0 10.0.0.1
    fs-nfs3 10.0.0.2:/armle/ / 10.0.0.2:/etc /etc &
#  Wait for a "bin" for the rest of the commands
    waitfor /usr/bin

# Some common servers
    pipe &
    mqueue &
    devc-pty &

    [+session] sh &
}

# make /tmp point to the shared memory area
[type=link] /tmp=/dev/shmem

# Redirect console messages
[type=link] /dev/console=/dev/ser1

# Programs require the runtime linker (ldqnx.so) to be at
# a fixed location
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
# We use the C shared lib (which also contains the runtime linker)
libc.so

# If some one needs floating point...
fpemu.so.2

# Include the network files so we can access files across the net
devn-tulip-abc123.so

# Include the socket library
libsocket.so
[data=copy]

# Include the network executables.
devc-ser8250
io-pkt-v4
fs-nfs3
```

For this release of Neutrino, you can't use the floating-point emulator (**fpemu.so.2**) in statically linked executables.

This buildfile is very similar to the previous one shown for the disk. The major difference is that instead of starting **devb-eide** to get a disk filesystem driver running, we started **io-pkt-v4** to get the network drivers running. The **-d** specifies the driver that should be loaded, in this case the driver for a DEC 21x4x (Tulip)-compatible Ethernet controller.

Once the network manager is running, we need to synchronize the script file interpretation to the availability of the drivers. That's what the **waitfor /dev/socket** is for — it waits for the network manager to initialize itself. The **ifconfig en0 10.0.0.1** command then specifies the IP address of the interface.

The next thing started is the NFS filesystem module, **fs-nfs3**, with options telling it that it should mount the filesystem present on **10.0.0.2** in two different places: **${QNX_TARGET}** should be mounted in **/**, and **/etc** should be mounted as **/etc**.

Since it may take some time to go over the network and establish the mounting, we see another **waitfor**, this time ensuring that the filesystem on the remote has been correctly mounted (here we assume that the remote has a directory called **${QNX_TARGET}/armle/bin** — since we've mounted the remote's **${QNX_TARGET}** as **/**, the **waitfor** is really waiting for **armle/bin** under the remote's **${QNX_TARGET}** to show up).

Processor-specific notes

In this section, we'll look at what's different from the generic files listed above for each processor family. Since almost everything that's processor- and platform-specific in Neutrino is contained in the kernel and startup programs, there's very little change required to go from an x86 with standard BIOS to, for example, a PowerPC 800 evaluation board.

Specifying the processor

The first obvious difference is that you must specify the processor that the buildfile is for. This is actually a simple change — in the **[virtual=...]** line, substitute the **x86** specification with **armle**, **mipsbe**, **ppcbe**, or **shle**.

Examples

For this CPU:	Use this attribute:
ARM (little-endian)	**[virtual=armle,binary]**
MIPS (big-endian)	**[virtual=mipsbe,elf]**

continued...

For this CPU:	Use this attribute:
PPC (big-endian)	`[virtual=ppcbe,openbios]`
SH-4 (little-endian)	`[virtual=shle,srec]`

Specifying the startup program

Another difference is that the startup program is tailored not only for the processor family, but also for the actual board the processor runs on. If you're not running an x86 with a standard BIOS, you should replace the **`startup-bios`** command with one of the many **`startup-*`** programs we supply.

To find out what startup programs we currently provide, please refer to the following sources:

- the **`boards`** directory under *bsp_working_dir*`/src/hardware/startup`
- QNX docs (BSP docs as well as **`startup-*`** entries in the *Utilities Reference*)
- the Community area of our website, **`www.qnx.com`**

Specifying the serial device

The examples listed previously provide support for the 8250 family of serial chips. Some non-x86 platforms support the 8250 family as well, but others have their own serial port chips.

For details on our current serial drivers, see:

- **`devc-*`** entries in the *Utilities Reference*
- the Community area of our website, **`www.qnx.com`**

Glossary

A20 gate

On x86-based systems, a hardware component that forces the A20 address line on the bus to zero, regardless of the actual setting of the A20 address line on the processor. This component is in place to support legacy systems, but the QNX Neutrino OS doesn't require any such hardware. Note that some processors, such as the 386EX, have the A20 gate hardware built right into the processor itself — our IPL will disable the A20 gate as soon as possible after startup.

adaptive

Scheduling algorithm whereby a thread's priority is decayed by 1. See also **FIFO**, **round robin**, and **sporadic**.

adaptive partitioning

A method of dividing, in a flexible manner, CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

asymmetric multiprocessing (AMP)

A multiprocessing system where a separate OS, or a separate instantiation of the same OS, runs on each CPU.

atomic

Of or relating to atoms. :-)

In operating systems, this refers to the requirement that an operation, or sequence of operations, be considered *indivisible*. For example, a thread may need to move a file position to a given location and read data. These operations must be performed in an atomic manner; otherwise, another thread could preempt the original thread and move the file position to a different location, thus causing the original thread to read data from the second thread's position.

attributes structure

Structure containing information used on a per-resource basis (as opposed to the **OCB**, which is used on a per-open basis).

This structure is also known as a **handle**. The structure definition is fixed (`iofunc_attr_t`), but may be extended. See also **mount structure**.

bank-switched

A term indicating that a certain memory component (usually the device holding an **image**) isn't entirely addressable by the processor. In this case, a hardware component manifests a small portion (or "window") of the device onto the processor's address bus. Special commands have to be issued to the hardware to move the window to different locations in the device. See also **linearly mapped**.

base layer calls

Convenient set of library calls for writing resource managers. These calls all start with *resmgr_*()*. Note that while some base layer calls are unavoidable (e.g. *resmgr_pathname_attach()*), we recommend that you use the **POSIX layer calls** where possible.

BIOS/ROM Monitor extension signature

A certain sequence of bytes indicating to the BIOS or ROM Monitor that the device is to be considered an "extension" to the BIOS or ROM Monitor — control is to be transferred to the device by the BIOS or ROM Monitor, with the expectation that the device will perform additional initializations.

On the x86 architecture, the two bytes 0x55 and 0xAA must be present (in that order) as the first two bytes in the device, with control being transferred to offset 0x0003.

block-integral

The requirement that data be transferred such that individual structure components are transferred in their entirety — no partial structure component transfers are allowed.

In a resource manager, directory data must be returned to a client as **block-integral** data. This means that only complete **`struct dirent`** structures can be returned — it's inappropriate to return partial structures, assuming that the next _IO_READ request will "pick up" where the previous one left off.

bootable

An image can be either bootable or **nonbootable**. A bootable image is one that contains the startup code that the IPL can transfer control to.

bootfile

The part of an OS image that runs the **startup code** and the Neutrino microkernel.

bound multiprocessing (BMP)

A multiprocessing system where a single instantiation of an OS manages all CPUs simultaneously, but you can lock individual applications or threads to a specific CPU.

budget

In **sporadic** scheduling, the amount of time a thread is permitted to execute at its normal priority before being dropped to its low priority.

buildfile

A text file containing instructions for **`mkifs`** specifying the contents and other details of an **image**, or for **`mkefs`** specifying the contents and other details of an embedded filesystem image.

canonical mode

Also called edited mode or "cooked" mode. In this mode the character device library performs line-editing operations on each received character. Only when a line is "completely entered" — typically when a carriage return (CR) is received — will the line of data be made available to application processes. Contrast **raw mode**.

channel

A kernel object used with message passing.

In QNX Neutrino, message passing is directed towards a **connection** (made to a channel); threads can receive messages from channels. A thread that wishes to receive messages creates a channel (using *ChannelCreate()*), and then receives messages from that channel (using *MsgReceive()*). Another thread that wishes to send a message to the first thread must make a connection to that channel by "attaching" to the channel (using *ConnectAttach()*) and then sending data (using *MsgSend()*).

chid

An abbreviation for **channel ID**.

CIFS

Common Internet File System (aka SMB) — a protocol that allows a client workstation to perform transparent file access over a network to a Windows 95/98/NT server. Client file access calls are converted to CIFS protocol requests and are sent to the server over the network. The server receives the request, performs the actual filesystem operation, and sends a response back to the client.

CIS

Card Information Structure — a data block that maintains information about flash configuration. The CIS description includes the types of memory devices in the regions, the physical geometry of these devices, and the partitions located on the flash.

coid

An abbreviation for **connection ID**.

combine message

A resource manager message that consists of two or more messages. The messages are constructed as combine messages by the client's C library (e.g. *stat()*, *readblock()*), and then handled as individual messages by the resource manager.

The purpose of combine messages is to conserve network bandwidth and/or to provide support for atomic operations. See also **connect message** and **I/O message**.

connect message

In a resource manager, a message issued by the client to perform an operation based on a pathname (e.g. an `io_open` message). Depending on the type of connect message sent, a context block (see **OCB**) may be associated with the request and will be passed to subsequent I/O messages. See also **combine message** and **I/O message**.

connection

A kernel object used with message passing.

Connections are created by client threads to "connect" to the channels made available by servers. Once connections are established, clients can *MsgSendv()* messages over them. If a number of threads in a process all attach to the same channel, then the one connection is shared among all the threads. Channels and connections are identified within a process by a small integer.

The key thing to note is that connections and file descriptors (**FD**) are one and the same object. See also **channel** and **FD**.

context

Information retained between invocations of functionality.

When using a resource manager, the client sets up an association or **context** within the resource manager by issuing an *open()* call and getting back a file descriptor. The resource manager is responsible for storing the information required by the context (see **OCB**). When the client issues further file-descriptor based messages, the resource manager uses the OCB to determine the context for interpretation of the client's messages.

cooked mode

See **canonical mode**.

core dump

A file describing the state of a process that terminated abnormally.

critical section

A code passage that *must* be executed "serially" (i.e. by only one thread at a time). The simplest from of critical section enforcement is via a **mutex**.

deadlock

A condition in which one or more threads are unable to continue due to resource contention. A common form of deadlock can occur when one thread sends a message to another, while the other thread sends a message to the first. Both threads are now waiting for each other to reply to the message. Deadlock can be avoided by good design practices or massive kludges — we recommend the good design approach.

device driver

A process that allows the OS and application programs to make use of the underlying hardware in a generic way (e.g. a disk drive, a network interface). Unlike OSs that require device drivers to be tightly bound into the OS itself, device drivers for QNX Neutrino are standard processes that can be started and stopped dynamically. As a result, adding device drivers doesn't affect any other part of the OS — drivers can be developed and debugged like any other application. Also, device drivers are in their own protected address space, so a bug in a device driver won't cause the entire OS to shut down.

discrete (or traditional) multiprocessor system

A system that has separate physical processors hooked up in multiprocessing mode over a board-level bus.

DNS

Domain Name Service — an Internet protocol used to convert ASCII domain names into IP addresses. In QNX native networking, **`dns`** is one of **Qnet**'s builtin resolvers.

dynamic bootfile

An OS image built on the fly. Contrast **static bootfile**.

dynamic linking

The process whereby you link your modules in such a way that the Process Manager will link them to the library modules before your program runs. The word "dynamic" here means that the association between your program and the library modules that it uses is done *at load time*, not at linktime. Contrast **static linking**. See also **runtime loading**.

edge-sensitive

One of two ways in which a **PIC** (Programmable Interrupt Controller) can be programmed to respond to interrupts. In edge-sensitive mode, the interrupt is "noticed" upon a transition to/from the rising/falling edge of a pulse. Contrast **level-sensitive**.

edited mode

See **canonical mode**.

EOI

End Of Interrupt — a command that the OS sends to the PIC after processing all Interrupt Service Routines (ISR) for that particular interrupt source so that the PIC can reset the processor's In Service Register. See also **PIC** and **ISR**.

EPROM

Erasable Programmable Read-Only Memory — a memory technology that allows the device to be programmed (typically with higher-than-operating voltages, e.g. 12V), with the characteristic that any bit (or bits) may be individually programmed from a 1 state to a 0 state. To change a bit from a 0 state into a 1 state can only be accomplished by erasing the *entire* device, setting *all* of the bits to a 1 state. Erasing is accomplished by shining an ultraviolet light through the erase window of the device for a fixed period of time (typically 10-20 minutes). The device is further characterized by having a limited number of erase cycles (typically 10e5 - 10e6). Contrast **flash** and **RAM**.

event

A notification scheme used to inform a thread that a particular condition has occurred. Events can be signals or pulses in the general case; they can also be unblocking events or interrupt events in the case of kernel timeouts and interrupt service routines. An event is delivered by a thread, a timer, the kernel, or an interrupt service routine when appropriate to the requestor of the event.

FD

File Descriptor — a client must open a file descriptor to a resource manager via the *open()* function call. The file descriptor then serves as a handle for the client to use in subsequent messages. Note that a file descriptor is the exact same object as a connection ID (*coid*, returned by *ConnectAttach()*).

FIFO

First In First Out — a scheduling algorithm whereby a thread is able to consume CPU at its priority level without bounds. See also **adaptive**, **round robin**, and **sporadic**.

flash memory

A memory technology similar in characteristics to **EPROM** memory, with the exception that erasing is performed electrically instead of via ultraviolet light, and, depending upon the organization of the flash memory device, erasing may be accomplished in blocks (typically 64k bytes at a time) instead of the entire device. Contrast **EPROM** and **RAM**.

FQNN

Fully Qualified NodeName — a unique name that identifies a QNX Neutrino node on a network. The FQNN consists of the nodename plus the node domain tacked together.

garbage collection

Aka space reclamation, the process whereby a filesystem manager recovers the space occupied by deleted files and directories.

HA

High Availability — in telecommunications and other industries, HA describes a system's ability to remain up and running without interruption for extended periods of time.

handle

A pointer that the resource manager base library binds to the pathname registered via *resmgr_attach()*. This handle is typically used to associate some kind of per-device information. Note that if you use the *iofunc_*()* **POSIX layer calls**, you must use a particular *type* of handle — in this case called an **attributes structure**.

hard thread affinity

A user-specified binding of a thread to a set of processors, done by means of a **runmask**. Contrast **soft thread affinity**.

image

In the context of embedded QNX Neutrino systems, an "image" can mean either a structure that contains files (i.e. an OS image) or a structure that can be used in a read-only, read/write, or read/write/reclaim FFS-2-compatible filesystem (i.e. a flash filesystem image).

inherit mask

A bitmask that specifies which processors a thread's children can run on. Contrast **runmask**.

interrupt

An event (usually caused by hardware) that interrupts whatever the processor was doing and asks it do something else. The hardware will generate an interrupt whenever it has reached some state where software intervention is required.

interrupt handler

See **ISR**.

interrupt latency

The amount of elapsed time between the generation of a hardware interrupt and the first instruction executed by the relevant interrupt service routine. Also designated as "T_{il}". Contrast **scheduling latency**.

interrupt service routine

See **ISR**.

interrupt service thread

A thread that is responsible for performing thread-level servicing of an interrupt.

Since an **ISR** can call only a very limited number of functions, and since the amount of time spent in an ISR should be kept to a minimum, generally the bulk of the interrupt servicing work should be done by a thread. The thread attaches the interrupt (via *InterruptAttach()* or *InterruptAttachEvent()*) and then blocks (via *InterruptWait()*), waiting for the ISR to tell it to do something (by returning an event of type SIGEV_INTR). To aid in minimizing **scheduling latency**, the interrupt service thread should raise its priority appropriately.

I/O message

A message that relies on an existing binding between the client and the resource manager. For example, an _IO_READ message depends on the client's having previously established an association (or **context**) with the resource manager by issuing an *open()* and getting back a file descriptor. See also **connect message**, **context**, **combine message**, and **message**.

I/O privileges

A particular right, that, if enabled for a given thread, allows the thread to perform I/O instructions (such as the x86 assembler **`in`** and **`out`** instructions). By default, I/O privileges are disabled, because a program with it enabled can wreak havoc on a system. To enable I/O privileges, the thread must be running as **`root`**, and call *ThreadCtl()*.

IPC

Interprocess Communication — the ability for two processes (or threads) to communicate. QNX Neutrino offers several forms of IPC, most notably native messaging (synchronous, client/server relationship), POSIX message queues and pipes (asynchronous), as well as signals.

IPL

Initial Program Loader — the software component that either takes control at the processor's reset vector (e.g. location 0xFFFFFFF0 on the x86), or is a BIOS extension. This component is responsible for setting up the machine into a usable state, such that the startup program can then perform further initializations. The IPL is written in assembler and C. See also **BIOS extension signature** and **startup code**.

IRQ

Interrupt Request — a hardware request line asserted by a peripheral to indicate that it requires servicing by software. The IRQ is handled by the **PIC**, which then interrupts the processor, usually causing the processor to execute an **Interrupt Service Routine (ISR)**.

ISR

Interrupt Service Routine — a routine responsible for servicing hardware (e.g. reading and/or writing some device ports), for updating some data structures shared between the ISR and the thread(s) running in the application, and for signalling the thread that some kind of event has occurred.

kernel

See **microkernel**.

level-sensitive

One of two ways in which a **PIC** (Programmable Interrupt Controller) can be programmed to respond to interrupts. If the PIC is operating in level-sensitive mode, the IRQ is considered active whenever the corresponding hardware line is active. Contrast **edge-sensitive**.

linearly mapped

A term indicating that a certain memory component is entirely addressable by the processor. Contrast **bank-switched**.

message

A parcel of bytes passed from one process to another. The OS attaches no special meaning to the content of a message — the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several processes. As they send, receive, and reply to messages, processes undergo various "changes of state" that affect when, and for how long, they may run.

microkernel

A part of the operating system that provides the minimal services used by a team of optional cooperating processes, which in turn provide the higher-level OS functionality. The microkernel itself lacks filesystems and many other services normally expected of an OS; those services are provided by optional processes.

mount structure

An optional, well-defined data structure (of type **`iofunc_mount_t`**) within an *iofunc_*()* structure, which contains information used on a per-mountpoint basis (generally used only for filesystem resource managers). See also **attributes structure** and **OCB**.

mountpoint

The location in the pathname space where a resource manager has "registered" itself. For example, the serial port resource manager registers mountpoints for each serial

device (**/dev/ser1**, **/dev/ser2**, etc.), and a CD-ROM filesystem may register a single mountpoint of **/cdrom**.

multicore system

A chip that has one physical processor with multiple CPUs interconnected over a chip-level bus.

mutex

Mutual exclusion lock, a simple synchronization service used to ensure exclusive access to data shared between threads. It is typically acquired (*pthread_mutex_lock()*) and released (*pthread_mutex_unlock()*) around the code that accesses the shared data (usually a **critical section**). See also **critical section**.

name resolution

In a QNX Neutrino network, the process by which the **Qnet** network manager converts an **FQNN** to a list of destination addresses that the transport layer knows how to get to.

name resolver

Program code that attempts to convert an **FQNN** to a destination address.

nd

An abbreviation for **node descriptor**, a numerical identifier for a node *relative to the current node*. Each node's node descriptor for itself is 0 (ND_LOCAL_NODE).

NDP

Node Discovery Protocol — proprietary QNX Software Systems protocol for broadcasting name resolution requests on a QNX Neutrino LAN.

network directory

A directory in the pathname space that's implemented by the **Qnet** network manager.

Neutrino

Name of an OS developed by QNX Software Systems.

NFS

Network FileSystem — a TCP/IP application that lets you graft remote filesystems (or portions of them) onto your local namespace. Directories on the remote systems appear as part of your local filesystem and all the utilities you use for listing and managing files (e.g. **ls**, **cp**, **mv**) operate on the remote files exactly as they do on your local files.

NMI

Nonmaskable Interrupt — an interrupt that can't be masked by the processor. We don't recommend using an NMI!

Node Discovery Protocol

See **NDP**.

node domain

A character string that the **Qnet** network manager tacks onto the nodename to form an **FQNN**.

nodename

A unique name consisting of a character string that identifies a node on a network.

nonbootable

A nonbootable OS image is usually provided for larger embedded systems or for small embedded systems where a separate, configuration-dependent setup may be required. Think of it as a second "filesystem" that has some additional files on it. Since it's nonbootable, it typically won't contain the OS, startup file, etc. Contrast **bootable**.

OCB

Open Control Block (or Open Context Block) — a block of data established by a resource manager during its handling of the client's *open()* function. This context block is bound by the resource manager to this particular request, and is then automatically passed to all subsequent I/O functions generated by the client on the file descriptor returned by the client's *open()*.

package filesystem

A virtual filesystem manager that presents a customized view of a set of files and directories to a client. The "real" files are present on some medium; the package filesystem presents a virtual view of selected files to the client.

partition

A division of CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

pathname prefix

See **mountpoint**.

pathname space mapping

The process whereby the Process Manager maintains an association between resource managers and entries in the pathname space.

persistent

When applied to storage media, the ability for the medium to retain information across a power-cycle. For example, a hard disk is a persistent storage medium, whereas a ramdisk is not, because the data is lost when power is lost.

Photon microGUI

The proprietary graphical user interface built by QNX Software Systems.

PIC

Programmable Interrupt Controller — hardware component that handles IRQs. See also **edge-sensitive**, **level-sensitive**, and **ISR**.

PID

Process ID. Also often *pid* (e.g. as an argument in a function call).

POSIX

An IEEE/ISO standard. The term is an acronym (of sorts) for Portable Operating System Interface — the "X" alludes to "UNIX", on which the interface is based.

POSIX layer calls

Convenient set of library calls for writing resource managers. The POSIX layer calls can handle even more of the common-case messages and functions than the **base layer calls**. These calls are identified by the *iofunc_*()* prefix. In order to use these (and we strongly recommend that you do), you must also use the well-defined POSIX-layer **attributes** (`iofunc_attr_t`), **OCB** (`iofunc_ocb_t`), and (optionally) **mount** (`iofunc_mount_t`) structures.

preemption

The act of suspending the execution of one thread and starting (or resuming) another. The suspended thread is said to have been "preempted" by the new thread. Whenever a lower-priority thread is actively consuming the CPU, and a higher-priority thread becomes READY, the lower-priority thread is immediately preempted by the higher-priority thread.

prefix tree

The internal representation used by the Process Manager to store the pathname table.

priority inheritance

The characteristic of a thread that causes its priority to be raised or lowered to that of the thread that sent it a message. Also used with mutexes. Priority inheritance is a method used to prevent **priority inversion**.

priority inversion

A condition that can occur when a low-priority thread consumes CPU at a higher priority than it should. This can be caused by not supporting priority inheritance, such that when the lower-priority thread sends a message to a higher-priority thread, the higher-priority thread consumes CPU *on behalf of* the lower-priority thread. This is solved by having the higher-priority thread inherit the priority of the thread on whose behalf it's working.

process

A nonschedulable entity, which defines the address space and a few data areas. A process must have at least one **thread** running in it — this thread is then called the first thread.

process group

A collection of processes that permits the signalling of related processes. Each process in the system is a member of a process group identified by a process group ID. A newly created process joins the process group of its creator.

process group ID

The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. The system may reuse a process group ID after the process group dies.

process group leader

A process whose ID is the same as its process group ID.

process ID (PID)

The unique identifier representing a process. A PID is a positive integer. The system may reuse a process ID after the process dies, provided no existing process group has the same ID. Only the Process Manager can have a process ID of 1.

pty

Pseudo-TTY — a character-based device that has two "ends": a master end and a slave end. Data written to the master end shows up on the slave end, and vice versa. These devices are typically used to interface between a program that expects a character device and another program that wishes to use that device (e.g. the shell and the `telnet` daemon process, used for logging in to a system over the Internet).

pulses

In addition to the synchronous Send/Receive/Reply services, QNX Neutrino also supports fixed-size, nonblocking messages known as pulses. These carry a small payload (four bytes of data plus a single byte code). A pulse is also one form of **event** that can be returned from an ISR or a timer. See *MsgDeliverEvent()* for more information.

Qnet

The native network manager in QNX Neutrino.

QoS

Quality of Service — a policy (e.g. **`loadbalance`**) used to connect nodes in a network in order to ensure highly dependable transmission. QoS is an issue that often arises in high-availability (**HA**) networks as well as realtime control systems.

RAM

Random Access Memory — a memory technology characterized by the ability to read and write any location in the device without limitation. Contrast **flash** and **EPROM**.

raw mode

In raw input mode, the character device library performs no editing on received characters. This reduces the processing done on each character to a minimum and provides the highest performance interface for reading data. Also, raw mode is used with devices that typically generate binary data — you don't want any translations of the raw binary stream between the device and the application. Contrast **canonical mode**.

replenishment

In **sporadic** scheduling, the period of time during which a thread is allowed to consume its execution **budget**.

reset vector

The address at which the processor begins executing instructions after the processor's reset line has been activated. On the x86, for example, this is the address 0xFFFFFFF0.

resource manager

A user-level server program that accepts messages from other programs and, optionally, communicates with hardware. QNX Neutrino resource managers are responsible for presenting an interface to various types of devices, whether actual (e.g. serial ports, parallel ports, network cards, disk drives) or virtual (e.g. **`/dev/null`**, a network filesystem, and pseudo-ttys).

In other operating systems, this functionality is traditionally associated with **device drivers**. But unlike device drivers, QNX Neutrino resource managers don't require any special arrangements with the kernel. In fact, a resource manager looks just like any other user-level program. See also **device driver**.

RMA

Rate Monotonic Analysis — a set of methods used to specify, analyze, and predict the timing behavior of realtime systems.

round robin

Scheduling algorithm whereby a thread is given a certain period of time to run. Should the thread consume CPU for the entire period of its timeslice, the thread will be placed at the end of the ready queue for its priority, and the next available thread will be made READY. If a thread is the only thread READY at its priority level, it will be able to consume CPU again immediately. See also **adaptive**, **FIFO**, and **sporadic**.

runmask

A bitmask that indicates which processors a thread can run on. Contrast **inherit mask**.

runtime loading

The process whereby a program decides *while it's actually running* that it wishes to load a particular function from a library. Contrast **static linking**.

scheduling latency

The amount of time that elapses between the point when one thread makes another thread READY and when the other thread actually gets some CPU time. Note that this latency is almost always at the control of the system designer.

Also designated as "T_{sl}". Contrast **interrupt latency**.

scoid

An abbreviation for **server connection ID**.

session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process belongs to the session that its process group belongs to. A newly created process joins the session of its creator. A process can alter its session membership via *setsid()*. A session can contain multiple process groups.

session leader

A process whose death causes all processes within its process group to receive a SIGHUP signal.

soft thread affinity

The scheme whereby the microkernel tries to dispatch a thread to the processor where it last ran, in an attempt to reduce thread migration from one processor to another, which can affect cache performance. Contrast **hard thread affinity**.

software interrupts

Similar to a hardware interrupt (see **interrupt**), except that the source of the interrupt is software.

sporadic

Scheduling algorithm whereby a thread's priority can oscillate dynamically between a "foreground" or normal priority and a "background" or low priority. A thread is given an execution **budget** of time to be consumed within a certain **replenishment** period. See also **adaptive**, **FIFO**, and **round robin**.

startup code

The software component that gains control after the IPL code has performed the minimum necessary amount of initialization. After gathering information about the system, the startup code transfers control to the OS.

static bootfile

An image created at one time and then transmitted whenever a node boots. Contrast **dynamic bootfile**.

static linking

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

symmetric multiprocessing (SMP)

A multiprocessor system where a single instantiation of an OS manages all CPUs simultaneously, and applications can float to any of them.

system page area

An area in the kernel that is filled by the startup code and contains information about the system (number of bytes of memory, location of serial ports, etc.) This is also called the SYSPAGE area.

thread

The schedulable entity under QNX Neutrino. A thread is a flow of execution; it exists within the context of a **process**.

tid

An abbreviation for **thread ID**.

timer

A kernel object used in conjunction with time-based functions. A timer is created via *timer_create()* and armed via *timer_settime()*. A timer can then deliver an **event**, either periodically or on a one-shot basis.

timeslice

A period of time assigned to a **round-robin** or **adaptive** scheduled thread. This period of time is small (on the order of tens of milliseconds); the actual value shouldn't be relied upon by any program (it's considered bad design).

Index

!

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z