Chapter 6

User Space Initialization

In This Chapter

In Chapter 2, "The Big Picture," we pointed out that the Linux kernel itself is but a small part of any embedded Linux system. After the kernel has initialized itself, it must mount a root file system and execute a set of developer-defined initialization routines. In this chapter, we examine the details of post-kernel system initialization.

We begin by looking at the root file system and its layout. Next we develop and study a minimal system configuration. Later in this chapter, we add functionality to the minimal system configuration to produce useful sample embedded system configurations. We complete the coverage of system initialization by introducing the initial ramdisk, `initrd` and `initramfs`, and its operation and use. The chapter concludes with a brief look at Linux shutdown logic.

6.1 Root File System

In Chapter 5, "Kernel Initialization," we examined the Linux kernel's behavior during the initialization process. We made several references to mounting a root file system. Linux, like many other advanced operating systems, requires a root file system to realize the benefits of its services. Although it is certainly possible to use Linux in an environment without a file system, doing so makes little sense, because most of the features and value of Linux would be lost. It would be similar to putting your entire system application into a bloated device driver or kernel thread. And can you imagine running your Windows PC without a file system?

The root file system refers to the file system mounted at the base of the file system hierarchy, designated simply as `/`. As you will discover in Chapter 9, "File Systems," even a small embedded Linux system typically mounts several file systems on different locations within the file system hierarchy. These include both real and virtual file systems such as `/proc` and `/sys`. The `proc` file system, introduced in Chapter 9, is an example. It is a special-purpose file system mounted at `/proc` under the root file system. The root file system is simply the first file system mounted at the top of the file system hierarchy.

As you will see shortly, the root file system has special requirements for a Linux system. Linux expects the root file system to contain programs and utilities to boot the system, initialize services such as networking and a system console, load device drivers, and mount additional file systems.

6.1.1 FHS: File System Hierarchy Standard

Several kernel developers authored a standard governing the organization and layout of a UNIX file system. The File System Hierarchy Standard (FHS) establishes a minimum baseline of compatibility between Linux distributions and application programs. You'll find a reference to this standard in the last section of this chapter. You are encouraged to review the FHS for a better background of the layout and rationale of UNIX file system organization.

Many Linux distributions have directory layouts closely matching that described in the FHS standard. The standard exists to provide one element of a common base between different UNIX and Linux distributions. The FHS standard allows your application software (and developers) to predict where certain system elements, including files and directories, can be found in the file system.

6.1.2 File System Layout

Where space is a concern, many embedded systems developers create a very small root file system on a bootable device (such as Flash memory). Later, a larger file system is mounted from another device, perhaps a hard disk or network file system (NFS) server. In fact, it is not uncommon to mount a larger root file system on top of the original small one. You'll see an example of that when we examine the initial ramdisk (`initrd` and `initramfs`) later in this chapter.

A simple Linux root file system might contain the following top-level directory entries:

```
 .
|
|--bin
|--dev
|--etc
|--home
|--lib
|--sbin
|--usr
|--var
|--tmp
```

Table 6-1 lists the most common contents of each of these root directory entries.

TABLE 6-1 Top-Level Directories

Directory	Contents
`bin`	Binary executables, usable by all users on the system[1]
`dev`	Device nodes (see Chapter 8, "Device Driver Basics")
`etc`	Local system configuration files
`home`	User account files
`lib`	System libraries, such as the standard C library and many others
`sbin`	Binary executables usually reserved for superuser accounts on the system
`tmp`	Temporary files
`usr`	A secondary file system hierarchy for application programs, usually read-only
`var`	Contains variable files, such as system logs and temporary configuration files

The very top of the Linux file system hierarchy is referenced by the slash character (`/`) by itself. For example, to list the contents of the root directory, you would type this:

```
$ ls /
```

This produces a listing similar to the following:

```
root@coyote:/# ls /
bin  dev  etc  home  lib  mnt  opt  proc  root  sbin  tmp  usr  var
root@coyote:/#
```

This directory listing contains directory entries for additional functionality, including `/mnt` and `/proc`. As previously noted, `/proc` is a special file system containing system information, and `/mnt` is a placeholder for user-mounted devices and file systems. Notice that we reference these directory entries preceded by a slash, indicating that the path to these top-level directories starts from the root directory.

6.1.3 Minimal File System

To illustrate the requirements of the root file system, we have created a minimal root file system. This example was produced on the ADI Engineering Coyote Reference board using an XScale processor. Listing 6-1 is the output from the `tree` command on this minimal root file system.

[1] Often embedded systems do not have user accounts other than a single root user.

LISTING 6-1 Contents of a Minimal Root File System

```
.
|-- bin
|   |-- busybox
|   `-- sh -> busybox
|-- dev
|   `-- console
|-- etc
|   `-- init.d
|       `-- rcS
`-- lib
    |-- ld-2.3.2.so
    |-- ld-linux.so.2 -> ld-2.3.2.so
    |-- libc-2.3.2.so
    `-- libc.so.6 -> libc-2.3.2.so

5 directories, 8 files
```

This root configuration makes use of `busybox`, a popular and aptly named toolkit for embedded systems. In short, `busybox` is a stand-alone binary that supports many common Linux command-line utilities. `busybox` is so pertinent for embedded systems that we devote Chapter 11, "BusyBox," to this flexible utility.

Notice that our sample minimum file system in Listing 6-1 has only eight files in five directories. This tiny root file system boots and provides the user with a fully functional command prompt on the serial console. Any commands that have been enabled in `busybox` are available to the user.

Starting from `/bin`, we have the `busybox` executable and a soft link called `sh` pointing back to `busybox`. You will see shortly why this is necessary. The file in `/dev` is a device node[2] required to open a console device for input and output. Although it is not strictly necessary, the `rcS` file in the `/etc/init.d` directory is the default initialization script processed by `busybox` on startup. Including `rcS` silences the warning message issued by `busybox` whenever `rcS` is missing.

The final directory entry and set of files required are the two libraries, glibc (`libc-2.3.2.so`) and the Linux dynamic loader (`ld-2.3.2.so`). glibc contains the standard C library functions, such as `printf()` and many others that most application programs depend on. The Linux dynamic loader is responsible for loading the binary executable into memory and performing the dynamic linking required by the application's reference to shared library functions. Two additional soft links are included—`ld-linux`.

[2] Device nodes are explained in detail in Chapter 8.

`so.2`, pointing back to `ld-2.3.2.so`, and `libc.so.6`, referencing `libc-2.3.2.so`. These links provide version immunity and backward compatibility for the libraries themselves and are found on all Linux systems.

This simple root file system produces a fully functional system. On the ARM/XScale board on which this was tested, the size of this small root file system was about 1.7MB. It is interesting to note that more than 80 percent of that size is contained within the C library itself. If you need to reduce its size for your embedded system, you might want to investigate the Library Optimizer Tool at http://libraryopt.sourceforge.net/.

6.1.4 The Embedded Root FS Challenge

The challenge of a root file system for an embedded device is simple to explain but not so simple to overcome. Unless you are lucky enough to be developing an embedded system with a reasonably large hard drive or large Flash storage on board, you might find it difficult to fit your applications and utilities onto a single Flash memory device. Although costs continue to come down for Flash storage, there will always be competitive pressure to reduce costs and decrease time to market. One of the single largest reasons Linux continues to grow in popularity as an embedded OS is the huge and growing body of Linux application software.

Trimming a root file system to fit into a given storage space requirement can be daunting. Many packages and subsystems consist of dozens or even hundreds of files. In addition to the application itself, many packages include configuration files, libraries, configuration utilities, icons, documentation files, locale files related to internationalization, database files, and more. The Apache web server from the Apache Software Foundation is an example of a well-known application often found in embedded systems. The base Apache package from one popular embedded Linux distribution contains 254 different files. Furthermore, they aren't all simply copied into a single directory on your file system. They need to be populated in several different locations on the file system for the Apache application to function without modification.

These concepts are some of the fundamental aspects of distribution engineering, and they can be quite tedious. Linux distribution companies such as Red Hat (in the desktop and enterprise market segments) and Mentor Graphics (in the embedded market segment) spend considerable engineering resources on just this: packaging a collection of programs, libraries, tools, utilities, and applications that together make up a Linux distribution. By necessity, building a root file system employs elements of distribution engineering on a smaller scale.

6.1.5 Trial-and-Error Method

Until recently, the only way to populate the contents of your root file system was to use the trial-and-error method. Perhaps the process could be automated by creating a set of scripts for this purpose, but the knowledge of which files are required for a given functionality still must come from the developer. Tools such as Red Hat Package Manager (`rpm`) can be used to install packages on your root file system. `rpm` has reasonable dependency resolution within given packages, but it is complex and involves a significant learning curve. Furthermore, using `rpm` does not lend itself easily to building small root file systems. It has limited capabilities for stripping unnecessary files from the installation, such as documentation and unused utilities for a given package.

6.1.6 Automated File System Build Tools

The leading vendors of embedded Linux distributions ship very capable tools designed to automate the task of building root file systems in Flash or other devices. These tools usually are graphical in nature, enabling the developer to select files by application or functionality. They have features to strip unnecessary files such as documentation from a package. Many let you select at the individual file level. These tools can produce a variety of file system formats for later installation on your choice of device. Contact your favorite embedded Linux distribution vendor for details on these powerful tools.

Some open source build tools automate the task of building a working root file system. Some of the more notable include `bitbake` from the OpenEmbedded project (www.openembedded.org/) and buildroot (http://buildroot.uclibc.org/.) Chapter 16, "Open Source Build Systems," presents details of some popular build systems.

6.2 Kernel's Last Boot Steps

The preceding chapter introduced the steps the kernel takes in the final phases of system boot. The final snippet of code from `.../init/main.c` is reproduced in Listing 6-2 for your convenience.

LISTING 6-2 Final Boot Steps from `main.c`

```
...
  if (execute_command) {
          run_init_process(execute_command);
          printk(KERN_WARNING "Failed to execute %s.  Attempting "
                              "defaults...\n", execute_command);
```

LISTING 6-2 Continued

```
    }

    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found.  Try passing init= option to kernel.");
```

This is the final sequence of events for the kernel thread called `kernel_init` spawned by the kernel during the final stages of boot. The `run_init_process()` is a small wrapper around the `execve()` function, which is a kernel system call with rather interesting behavior. The `execve()` function never returns if no error conditions are encountered in the call. The memory space in which the calling thread executes is overwritten by the called program's memory image. In effect, the called program directly replaces the calling thread, including inheriting its Process ID (PID).

The basic structure of this initialization sequence has been unchanged for a long time in the development of the Linux kernel. In fact, Linux version 1.0 contained similar constructs. Essentially, this is the start of user space[3] processing. As you can see from Listing 6-2, unless the Linux kernel is successful in executing one of these processes, the kernel will halt, displaying the message passed in the `panic()` system call. If you have been working with embedded systems for any length of time, and especially if you have experience working on root file systems, you are more than familiar with this kernel `panic()` and its message. If you do an Internet search for this `panic()` error message, you will find page after page of hits. When you complete this chapter, you will be an expert at troubleshooting this common failure.

Notice a key ingredient of these processes: They are all programs that are expected to reside on a root file system that has a structure similar to that presented in Listing 6-1. Therefore, we know that we must at least satisfy the kernel's requirement for an `init` process that can execute within its own environment.

In looking at Listing 6-2, this means that at least one of the `run_init_process()` function calls must succeed. You can see that the kernel tries to execute one of four programs in the order in which they are encountered. As you also can see that if none of these four programs succeeds, the booting kernel issues the dreaded `panic()` function call and dies right there. Remember, this snippet of code from `.../init/main.c`

[3] In actuality, modern Linux kernels create a userspace-like environment earlier in the boot sequence for specialized activities, which are beyond the scope of this book.

is executed only once on bootup. If it does not succeed, the kernel can do little but complain and halt, which it does through the `panic()` function call.

6.2.1 First User Space Program

On most Linux systems, `/sbin/init` is spawned by the kernel on boot. This is why it is attempted first from Listing 6-2. Effectively, this becomes the first user space program to run. To review, this is the sequence:

1. Mount the root file system.
2. Spawn the first user space program, which, in this discussion, becomes `/sbin/init`.

In our sample minimal root file system from Listing 6-1, the first three attempts at spawning a user space process would fail, because we did not provide an executable file called `init` anywhere on the file system. Recall from Listing 6-1 that we had a soft link called `sh` that pointed back to `busybox`. You should now realize the purpose of that soft link: It causes `busybox` to be executed by the kernel as the initial process while also satisfying the common requirement for a shell executable from user space.[4]

6.2.2 Resolving Dependencies

It is not sufficient to simply include an executable such as `init` on your file system and expect it to boot. For every process you place on your root file system, you must also satisfy its dependencies. Most processes have two categories of dependencies: those that are needed to satisfy unresolved references within a dynamically linked executable, and external configuration or data files that an application might need. We have a tool to find the former, but the latter can be supplied only by at least a cursory understanding of the application in question.

An example will help make this clear. The `init` process is a dynamically linked executable. To run `init`, we need to satisfy its library dependencies. A tool has been developed for this purpose: `ldd`. To understand what libraries a given application requires, simply run your cross-version of `ldd` on the binary:

```
$ ppc_4xx-ldd init
        libc.so.6 => /opt/eldk/ppc_4xxFP/lib/libc.so.6
        ld.so.1 => /opt/eldk/ppc_4xxFP/lib/ld.so.1
$
```

[4] When busybox is invoked via the `sh` symbolic link, it spawns a shell. We cover this in detail in Chapter 11.

From this `ldd` output, we can see that the Power Architecture `init` executable in this example is dependent on two libraries—the standard C library (`libc.so.6`) and the Linux dynamic loader (`ld.so.1`).

To satisfy the second category of dependencies for an executable, the configuration and data files that it might need, there is little substitute for some knowledge about how the subsystem works. For example, `init` expects to read its operational configuration from a data file called `inittab` located on `/etc`. Unless you are using a tool that has this knowledge built in, such as those described in Chapter 16, you must supply that knowledge.

6.2.3 Customized Initial Process

It is worth noting that the system user can control which initial process is executed at startup. This is done by a kernel command-line parameter. It is hinted at in Listing 6-2 by the text contained within the `panic()` function call. Building on our kernel command line from Chapter 5, here is how it might look with a user-specified `init` process:

```
console=ttyS0,115200 ip=bootp root=/dev/nfs init=/sbin/myinit
```

Specifying `init=` in the kernel command line in this way, you must provide a binary executable on your root file system in the `/sbin` directory called `myinit`. This would be the first process to gain control at the completion of the kernel's boot process.

6.3 The `init` Process

Unless you are doing something highly unusual, you will never need to provide a customized initial process, because the capabilities of the standard `init` process are very flexible. The `init` program, together with a family of startup scripts that we will examine shortly, implement what is commonly called System V Init, from the original UNIX System V that used this schema. We will now examine this powerful system configuration and control utility.

You saw in the preceding section that `init` is the first user space process spawned by the kernel after completion of the boot process. As you will learn, every process in a running Linux system has a child-parent relationship with another process running in the system. `init` is the ultimate parent of all user space processes in a Linux system. Furthermore, `init` provides the default set of environment parameters for all other processes to inherit, such as the initial system `PATH`.

Its primary role is to spawn additional processes under the direction of a special configuration file. This configuration file is usually stored as /etc/inittab. init has the concept of a runlevel. A runlevel can be thought of as a system state. Each runlevel is defined by the services that are enabled and programs that are spawned upon entry to that runlevel.

init can exist in a single runlevel at any given time. Runlevels used by init include runlevels from 0 to 6 and a special runlevel called S. Runlevel 0 instructs init to halt the system, and runlevel 6 results in a system reboot. For each runlevel, a set of startup and shutdown scripts is usually provided that define the action a system should take for each runlevel. Actions to perform for a given runlevel are determined by the /etc/inittab configuration file, described shortly.

Several of the runlevels have been reserved for specific purposes in many distributions. Table 6-2 describes the runlevels and their purposes in common use in many Linux distributions.

TABLE 6-2 Runlevels

Runlevel	Purpose
0	System shutdown (halt)
1	Single-user system configuration for maintenance
2	User-defined
3	General-purpose multiuser configuration
4	User-defined
5	Multiuser with graphical user interface on startup
6	System restart (reboot)

The runlevel scripts are commonly found under a directory called /etc/rc.d/init.d. Here you will find most of the scripts that enable and disable individual services. Services can be configured manually by invoking the script and passing one of the appropriate arguments to the script, such as start, stop, or restart. Listing 6-3 displays an example of restarting the NFS service.

LISTING 6-3 NFS Restart

```
$ /etc/init.d/nfs-kernel-server
Shutting down NFS mountd:                                  [  OK  ]
Shutting down NFS daemon:                                  [  OK  ]
Shutting down NFS quotas:                                  [  OK  ]
Shutting down NFS services:                                [  OK  ]
```

LISTING 6-3 Continued

```
Starting NFS services:                                     [  OK  ]
Starting NFS quotas:                                       [  OK  ]
Starting NFS daemon:                                       [  OK  ]
Starting NFS mountd:                                       [  OK  ]
```

If you have spent any time with a desktop Linux distribution such as Red Hat or Fedora, you have undoubtedly seen lines like this during system startup.

A runlevel is defined by the services that are enabled at that runlevel. Most Linux distributions contain a directory structure under `/etc` that contains symbolic links to the service scripts in `/etc/rc.d/init.d`. These runlevel directories typically are rooted at `/etc/rc.d`. Under this directory, you will find a series of runlevel directories that contain startup and shutdown specifications for each runlevel. `init` simply executes these scripts upon entry and exit from a runlevel. The scripts define the system state, and `inittab` instructs `init` which scripts to associate with a given runlevel. Listing 6-4 contains the directory structure beneath `/etc/rc.d` that drives the runlevel startup and shutdown behavior upon entry to or exit from the specified runlevel, respectively.

LISTING 6-4 Runlevel Directory Structure

```
$ ls -l /etc/rc.d
total 96
drwxr-xr-x  2 root root  4096 Oct 20 10:19 init.d
-rwxr-xr-x  1 root root  2352 Mar 16  2009 rc
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc0.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc1.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc2.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc3.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc4.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc5.d
drwxr-xr-x  2 root root  4096 Mar 22  2009 rc6.d
-rwxr-xr-x  1 root root   943 Dec 31 16:36 rc.local
-rwxr-xr-x  1 root root 25509 Jan 11  2009 rc.sysinit
```

Each of the runlevels is defined by the scripts contained in `rcN.d`, where *N* is the runlevel. Inside each `rcN.d` directory, you will find numerous symlinks arranged in a specific order. These symbolic links start with either a `K` or an `S`. Those beginning with `S` point to service scripts, which are invoked with startup instructions. Those starting with `K` point to service scripts that are invoked with shutdown instructions. An example with a very small number of services might look like Listing 6-5.

LISTING 6-5 Sample Runlevel Directory

```
lrwxrwxrwx  1 root root 17 Nov 25  2009 S10network -> ../init.d/network
lrwxrwxrwx  1 root root 16 Nov 25  2009 S12syslog  -> ../init.d/syslog
lrwxrwxrwx  1 root root 16 Nov 25  2009 S56xinetd  -> ../init.d/xinetd
lrwxrwxrwx  1 root root 16 Nov 25  2009 K50xinetd  -> ../init.d/xinetd
lrwxrwxrwx  1 root root 16 Nov 25  2009 K88syslog  -> ../init.d/syslog
lrwxrwxrwx  1 root root 17 Nov 25  2009 K90network -> ../init.d/network
```

This code instructs the startup scripts to start three services upon entry to this fictitious runlevel: `network`, `syslog`, and `xinetd`. Because the `S*` scripts are ordered with a numeric tag, they will be started in this order. In a similar fashion, when exiting this runlevel, three services will be terminated: `xinetd`, `syslog`, and `network`. In a similar fashion, these services will be terminated in the order presented by the two-digit number following the `K` in the symlink filename. In an actual system, there would undoubtedly be many more entries. You can include your own entries for your own custom applications as well.

The top-level script that executes these service startup and shutdown scripts is defined in the `init` configuration file, which we now examine.

6.3.1 inittab

When `init` is started, it reads the system configuration file `/etc/inittab`. This file contains directives for each runlevel, as well as directives that apply to all runlevels. This file and `init`'s behavior are well documented in man pages on most Linux workstations, as well as by several books covering system administration. We do not attempt to duplicate those works; we focus on how a developer might configure `inittab` for an embedded system. For a detailed explanation of how `inittab` and `init` work together, view the man page on most Linux workstations by typing `man init` and `man inittab`.

Let's look at a typical `inittab` for a simple embedded system. Listing 6-6 contains a simple `inittab` example for a system that supports a single runlevel as well as shutdown and reboot.

LISTING 6-6 Simple `inittab`

```
# /etc/inittab

# The default runlevel (2 in this example)
id:2:initdefault:
```

LISTING 6-6 Continued

```
# This is the first process (actually a script) to be run.
si::sysinit:/etc/rc.sysinit

# Execute our shutdown script on entry to runlevel 0
l0:0:wait:/etc/init.d/sys.shutdown

# Execute our normal startup script on entering runlevel 2
l2:2:wait:/etc/init.d/runlvl2.startup

# This line executes a reboot script (runlevel 6)
l6:6:wait:/etc/init.d/sys.reboot

# This entry spawns a login shell on the console
# Respawn means it will be restarted each time it is killed
con:2:respawn:/bin/sh
```

This very simple[5] `inittab` script describes three individual runlevels. Each runlevel is associated with a script, which must be created by the developer for the desired actions in each runlevel. When this file is read by `init`, the first script to be executed is `/etc/rc.sysinit`. This is denoted by the `sysinit` tag. Then `init` enters runlevel 2 and executes the script defined for runlevel 2. From this example, this would be `/etc/init.d/runlvl2.startup`. As you might guess from the `:wait:` tag shown in Listing 6-6, `init` waits until the script completes before continuing. When the runlevel 2 script completes, `init` spawns a shell on the console (through the `/bin/sh` symbolic link), as shown in the last line of Listing 6-6. The `respawn` keyword instructs `init` to restart the shell each time it detects that it has exited. Listing 6-7 shows what it looks like during boot.

LISTING 6-7 Sample Startup Messages

```
...
VFS: Mounted root (nfs filesystem).
Freeing init memory: 304K
INIT: version 2.78 booting
This is rc.sysinit
INIT: Entering runlevel: 2
This is runlvl2.startup

#
```

[5] This `inittab` is a nice example of a small, purpose-built embedded system.

The startup scripts in this example do nothing except announce themselves for illustrative purposes. Of course, in an actual system, these scripts enable features and services that do useful work! Given the simple configuration in this example, you would enable the services and applications for your particular widget in the `/etc/init.d/runlvl2.startup` script. You would do the reverse—disable your applications, services, and devices—in your shutdown and/or reboot scripts. The next section looks at some typical system configurations and the required entries in the startup scripts to enable these configurations.

6.3.2 Sample Web Server Startup Script

Although simple, this sample startup script is designed to illustrate the mechanism and guide you in designing your own system startup and shutdown behavior. This example is based on `busybox`, which has a slightly different initialization behavior than `init`. These differences are covered in detail in Chapter 11.

In a typical embedded appliance that contains a web server, you might want several servers available for maintenance and remote access. In this example, we enable servers for HTTP and Telnet access (via `inetd`). Listing 6-8 contains a simple `rc.sysinit` script for our hypothetical web server appliance.

LISTING 6-8 Web Server `rc.sysinit`

```
#!/bin/sh

echo "This is rc.sysinit"

busybox mount -t proc none /proc

# Load the system loggers
/sbin/syslogd
/sbin/klogd

# Enable legacy PTY support for telnetd
busybox mkdir /dev/pts
busybox mknod /dev/ptmx c 5 2
busybox mount -t devpts devpts /dev/pts
```

In this simple initialization script, we first enable the proc file system. The details of this useful subsystem are covered in Chapter 9. Next we enable the system loggers so that we can capture system information during operation. This is especially useful

when things go wrong. The last entries enable support for the UNIX PTY subsystem, which is required for the implementation of the Telnet server used for this example.

Listing 6-9 contains the commands in the runlevel 2 startup script. This script contains the commands to enable any services we want to have operational for our appliance.

LISTING 6-9 Sample Runlevel 2 Startup Script

```
#!/bin/sh

echo "This is runlvl2.startup"

echo "Starting Internet Superserver"
inetd

echo "Starting web server"
webs &
```

Notice how simple this runlevel 2 startup script is. First we enable the so-called Internet superserver `inetd`, which intercepts and spawns services for common TCP/IP requests. In our example, we enabled Telnet services through a configuration file called `/etc/inetd.conf`. Then we execute the web server, here called `webs`. That's all there is to it. Although minimal, this is a working configuration for Telnet and web services.

To complete this configuration, you might supply a shutdown script (refer to Listing 6-6), which, in this case, would terminate the web server and the Internet superserver before system shutdown. In our sample scenario, that is sufficient for a clean shutdown.

6.4 Initial RAM Disk

The Linux kernel contains two mechanisms to mount an early root file system to perform certain startup-related system initialization and configuration. First we will discuss the legacy method, the initial ramdisk, or `initrd`. The next section covers the newer method called `initramfs`.

The legacy method for enabling early user space processing is known as the initial RAM disk, or simply `initrd`. Support for this functionality must be compiled into the kernel. This kernel configuration option is found under General Setup, RAM disk support in the kernel configuration utility. Figure 6-1 shows an example of the configuration for `initrd` and `initramfs`.

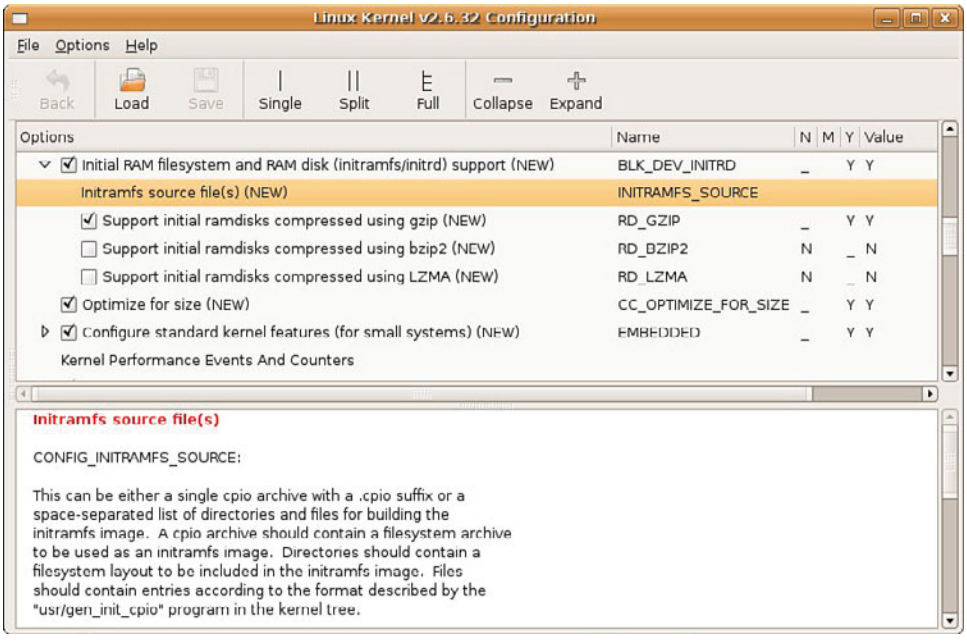

FIGURE 6-1 Linux kernel configuration utility

The initial RAM disk is a small, self-contained root file system that usually contains directives to load specific device drivers before the completion of the boot cycle. In Linux workstation distributions such as Red Hat and Ubuntu, an initial RAM disk is used to load the device drivers for the EXT3 file system before mounting the real root file system. An `initrd` is frequently used to load a device driver that is required in order to access the real root file system.

6.4.1 Booting with `initrd`

To use the `initrd` functionality, the bootloader gets involved on most architectures to pass the `initrd` image to the kernel. A common scenario is that the bootloader loads a compressed kernel image into memory and then loads an `initrd` image into another section of available memory. In doing so, it becomes the bootloader's responsibility to pass the load address of the `initrd` image to the kernel before passing control to it. The exact mechanism differs depending on the architecture, bootloader, and platform implementation. However, the kernel must know where the `initrd` image is located so it can load it.

Some architectures and platforms construct a single composite binary image. This scheme is used when the bootloader does not have specific Linux support for loading `initrd` images. In this case, the kernel and `initrd` image are simply concatenated into a single composite image. You will find reference to this type of composite image in the kernel makefiles as `bootpImage`. Presently, this is used only for the ARM architecture.[6]

So how does the kernel know where to find the `initrd` image? Unless there is some special magic in the bootloader, it is usually sufficient simply to pass the `initrd` image start address and size to the kernel via the kernel command line. Here is an example of a kernel command line for a popular ARM-based reference board containing the TI OMAP 5912 processor:

```
console=ttyS0,115200 root=/dev/nfs                          \
   nfsroot=192.168.1.9:/home/chris/sandbox/omap-target      \
   initrd=0x10800000,0x14af47
```

This kernel command line has been separated into several lines to fit in the space provided. In actual practice, it is a single line, with the individual elements separated by spaces. This kernel command line defines the following kernel behavior:

- Specify a single console on device `ttyS0` at 115 kilobaud.
- Mount a root file system via NFS, the network file system.
- Find the NFS root file system on host 192.168.1.9 (from directory `/home/chris/sandbox/omap-target`).
- Load and mount an initial ramdisk from physical memory location `0x10800000`, which has a size of `0x14AF47` (1,355,591 bytes).

One additional note regarding this example: Almost universally, the `initrd` image is compressed. The size specified on the kernel command line is the size of the compressed image.

6.4.2 Bootloader Support for `initrd`

Let's look at a simple example based on the popular U-Boot bootloader running on an ARM processor. This bootloader was designed with support for directly booting the Linux kernel. Using U-Boot, it is easy to include an `initrd` image with the kernel image. Listing 6-10 shows a typical boot sequence containing an initial ramdisk image.

[6] This technique has largely been deprecated in favor of using `initramfs`, as explained next.

LISTING 6-10 Booting the Kernel with Ramdisk Support

```
[uboot]> tftp 0x10000000 kernel-uImage
...
Load address: 0x10000000
Loading: ############################## done
Bytes transferred = 1069092 (105024 hex)

[uboot]> tftp 0x10800000 initrd-uboot
...
Load address: 0x10800000
Loading: ########################################## done
Bytes transferred = 282575 (44fcf hex)

[uboot]> bootm 0x10000000 0x10800040
Uncompressing kernel.................done.
...
RAMDISK driver initialized: 16 RAM disks of 16384K size 1024 blocksize
...
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 filesystem).
Greetings: this is linuxrc from Initial RAMDisk
Mounting /proc filesystem

BusyBox v1.00 (2005.03.14-16:37+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# (<<<< Busybox command prompt)
```

Here we get a glimpse of the U-Boot bootloader, which we'll examine in more detail in the next chapter. The `tftp` command causes U-Boot to download the kernel image from a TFTP server. The kernel image is downloaded and placed into the base of this target system's memory at the 256MB address (`0x10000000` hex[7]). Then a second image, the initial ramdisk image, is downloaded from a TFTP server into memory at a higher memory address (256MB + 8MB in this example). Finally, we issue the U-Boot `bootm` command, which is the "boot from memory" command. The `bootm` command takes two arguments: the address of the Linux kernel image, optionally followed by an address representing the location of the initial ramdisk image.

[7] It just so happens that on this particular board, our physical SDRAM starts at 256MB.

Take special note of one feature of the U-Boot bootloader: It fully supports loading kernel and ramdisk images over an Ethernet connection. This is a very useful development configuration. You can get a kernel and ramdisk image onto your board in other ways as well. You can flash them into your Flash memory using a hardware-based Flash programming tool, or you can use a serial port and download the kernel and file system images via RS-232. However, because these images typically are large (a kernel can be about a megabyte, and a ramdisk can be tens of megabytes), you will save a significant amount of engineering time if you invest in this Ethernet-based TFTP download method. Whatever bootloader you choose, make sure it supports network download of development images.

6.4.3 `initrd` Magic: `linuxrc`

When the kernel boots, first it detects the presence of the `initrd` image. Then it copies the compressed binary file from the specified physical location in RAM into a proper kernel ramdisk and mounts it as the root file system. The magic of `initrd` comes from the contents of a special file within the `initrd` image. When the kernel mounts the initial ramdisk, it looks for a specific file called `linuxrc`. It treats this file as a script file and proceeds to execute the commands contained therein. This mechanism enables the system designer to specify the behavior of `initrd`. Listing 6-11 shows a sample `linuxrc` file.

LISTING 6-11 Sample `linuxrc` File

```
#!/bin/sh

echo 'Greetings: this is 'linuxrc' from Initial Ramdisk'
echo 'Mounting /proc filesystem'
mount -t proc /proc /proc

busybox sh
```

In practice, this file would contain directives required before we mount the real root file system. One example might be to load CompactFlash drivers to obtain a real root file system from a CompactFlash device. For the purposes of this example, we simply spawn a `busybox` shell and halt the boot process for examination. You can see the `#` command prompt from Listing 6-10 resulting from this `busybox` shell. If you were to type the `exit` command here, the kernel would continue its boot process until complete.

After the kernel copies the ramdisk from physical memory into a kernel ramdisk, it returns this physical memory to the available memory pool. You can think of this as transferring the `initrd` image from physical memory at the hard-coded address into the kernel's own virtual memory (in the form of a kernel ramdisk device).

One last comment about Listing 6-11: The `mount` command in which the `/proc` file system is mounted seems redundant in its use of the word `proc`. This command would also work:

```
mount -t proc none /proc
```

Notice that the `device` field of the `mount` command has been changed to `none`. The `mount` command ignores the `device` field because no physical device is associated with the `proc` file system. The `-t proc` is enough to instruct `mount` to mount the `/proc` file system on the `/proc` mount point. I use the former invocation as a mental reminder that we are actually mounting the kernel pseudo device (the `/proc` file system) on `/proc`. The `mount` command ignores this argument. Use the method you prefer. Later, when you type `mount` at the command line, the device field will show `proc` instead of `none`, reminding you that this is a virtual file system.

6.4.4 The `initrd` Plumbing

As part of the Linux boot process, the kernel must locate and mount a root file system. Late in the boot process, the kernel decides what and where to mount in a function called `prepare_namespace()`, which is found in `.../init/do_mounts.c`. If `initrd` support is enabled in the kernel, as illustrated in Figure 6-1, and the kernel command line is so configured, the kernel decompresses the compressed `initrd` image from physical memory and eventually copies the contents of this file into a ramdisk device (`/dev/ram`). At this point, we have a proper file system on a kernel ramdisk. After the file system has been read into the ramdisk, the kernel effectively mounts this ramdisk device as its root file system. Finally, the kernel spawns a kernel thread to execute the `linuxrc` file on the `initrd` image.[8]

When the `linuxrc` script has completed execution, the kernel unmounts the `initrd` and proceeds with the final stages of system boot. If the real root device has a directory called `/initrd`, Linux mounts the `initrd` file system on this path (in this context, called

[8] Out of necessity (space), this is a very simplified description of the sequence of events. The actual mechanism is similar in concept, but several significant details are omitted for clarity. You are encouraged to consult the kernel source code for more details. See `.../init/main.c` and `.../init/do_mounts*.c`.

a mount point). If this directory does not exist in the final root file system, the `initrd` image is simply discarded.

If the kernel command line contains a `root=` parameter specifying a ramdisk (`root=/dev/ram0`, for example), the previously described `initrd` behavior changes in two important ways. First, the processing of the `linuxrc` executable is skipped. Second, no attempt is made to mount another file system as root. This means that you can have a Linux system with `initrd` as the only root file system. This is useful for minimal system configurations in which the only root file system is the ramdisk. Placing `/dev/ram0` on the kernel command line allows the full system initialization to complete with `initrd` as the final root file system.

6.4.5 Building an `initrd` Image

Constructing a suitable root file system image is one of the more challenging aspects of embedded systems. Creating a proper `initrd` image is even more challenging, because it needs to be small and specialized. This section examines `initrd` requirements and file system contents.

Listing 6-12 was produced by running the `tree` utility on our sample `initrd` image from this chapter.

LISTING 6-12 Contents of a Sample `initrd`

```
.
|-- bin
|   |-- busybox
|   |-- echo -> busybox
|   |-- mount -> busybox
|   `-- sh -> busybox
|-- dev
|   |-- console
|   |-- ram0
|   `-- ttyS0
|-- etc
|-- linuxrc
`-- proc

4 directories, 8 files
```

As you can see, it is very small indeed; it takes up a little more than 500KB in uncompressed form. Since it is based on `busybox`, it has many capabilities. Because `busybox` is

statically linked for this exercise, it has no dependencies on any system libraries. You will learn more about busybox in Chapter 11.

6.5 Using initramfs

initramfs is the preferred mechanism for executing early user space programs. It is conceptually similar to initrd, as described in the preceding section. It is enabled using the same configuration selections as shown in Figure 6-1. Its purpose is also similar: to enable loading of drivers that might be required before mounting the real (final) root file system. However, it differs in significant ways from the initrd mechanism.

The technical implementation details differ significantly between initrd and initramfs. For example, initramfs is loaded before the call to do_basic_setup(),[9] which provides a mechanism for loading firmware for devices before its driver has been loaded. For more details, see the Linux kernel documentation for this subsystem at .../Documentation/filesystems/ramfs-rootfs-initramfs.txt.

From a practical perspective, initramfs is much easier to use. initramfs is a cpio archive, whereas initrd is a gzipped file system image. This simple difference contributes to the ease of use of initramfs and removes the requirement that you must be root to create it. It is integrated into the Linux kernel source tree, and a small default (nearly empty) image is built automatically when you build the kernel image. Making changes to it is far easier than building and loading a new initrd image.

Listing 6-13 shows the contents of the Linux kernel .../usr directory, where the initramfs image is built. The contents of Listing 6-13 are shown after a kernel has been built.

LISTING 6-13 Kernel initramfs Build Directory

```
$ ls -l usr
total 72
-rw-r--r-- 1 chris chris  1146 2009-12-16 12:36 built-in.o
-rwxr-xr-x 1 chris chris 15567 2009-12-16 12:36 gen_init_cpio
-rw-r--r-- 1 chris chris 12543 2009-12-16 12:35 gen_init_cpio.c
-rw-r--r-- 1 chris chris  1024 2009-06-24 10:57 initramfs_data.bz2.S
-rw-r--r-- 1 chris chris   512 2009-12-16 12:36 initramfs_data.cpio
-rw-r--r-- 1 chris chris  1023 2009-06-24 10:57 initramfs_data.gz.S
-rw-r--r-- 1 chris chris  1025 2009-06-24 10:57 initramfs_data.lzma.S
```

[9] do_basic_setup is called from .../init/main.c and calls do_initcalls(). This causes driver module initialization routines to be called. This was described in detail in Chapter 5 and shown in Listing 5-10.

LISTING 6-13 Continued

```
-rw-r--r-- 1 chris chris  1158 2009-12-16 12:36 initramfs_data.o
-rw-r--r-- 1 chris chris  1021 2009-06-24 10:57 initramfs_data.S
-rw-r--r-- 1 chris chris  4514 2009-06-24 10:57 Kconfig
-rw-r--r-- 1 chris chris  2154 2009-12-16 12:35 Makefile
```

A build script in `.../scripts` called `gen_initramfs_list.sh` defines a default list of files that will be included in the `initramfs` archive. The default for recent Linux kernels looks like Listing 6-14.

LISTING 6-14 Sample `initramfs` File Specification

```
dir /dev 0755 0 0
nod /dev/console 0600 0 0 c 5 1
dir /root 0700 0 0
```

This produces a small default directory structure containing the `/root` and `/dev` top-level directories, as well as a single device node representing the console. The details of how to specify items for `initramfs` file systems are described in the kernel documentation at `.../Documentation/filesystems/ramfs-rootfs-initramfs.txt`. In summary, the preceding listing produces a directory entry (`dir`) called `/dev`, with 0755 file permissions and a user-id and group-id of 0 (root.) The second line defines a device node (`nod`) called `/dev/console`, with file permissions of 0600, user and group IDs of 0 (root), being a *character device* (`c`) with major number 5 and minor number 1.[10] The third line creates another directory called `/root` similar to the `/dev` specifier.

6.5.1 Customizing `initramfs`

There are two ways to customize the `initramfs` for your particular requirements. Either create a cpio archive with your required files, or specify a list of directories and files whose contents are merged with the default created by `gen_initramfs_list.sh`. You specify a source for your `initramfs` files via the kernel-configuration facility. Enable `INITRAMFS_SOURCE` in your kernel configuration, and point it to a location on your development workstation. This configuration parameter is highlighted in Figure 6-1. The kernel build system will use those files as the source for your `initramfs` image. Let's see what this looks like using a minimal file system similar to the one built in Listing 6-1.

[10] If you are unfamiliar with device nodes and the concept of major numbers and minor numbers, these topics are covered in Chapter 8.

First, we will build a file collection containing the files we want for a minimal system. Because initramfs is supposed to be small and lean, we'll build it around a statically compiled busybox. Compiling busybox statically means it is not dependent on any system libraries. We need very little beyond busybox: a device node for the console in a directory called /dev and a symlink pointing back to busybox called init. Finally, we'll include a busybox startup script to spawn a shell for us to interact with after booting into this initramfs. Listing 6-15 details this minimal file system.

LISTING 6-15 Minimal initramfs Contents

```
$ tree ./usr/myinitramfs_root/
.
|-- bin
|   |-- busybox
|   `-- sh -> busybox
|-- dev
|   `-- console
|-- etc
|   `-- init.d
|       `-- rcS
`-- init -> /bin/sh

4 directories, 5 files
```

When we point the kernel configuration parameter INITRAMFS_SOURCE to the directory where this file structure lives, it automatically builds the initramfs compressed cpio archive and links it into the kernel image.

The reason for the init symlink should be noted. When the kernel is configured for initramfs, it searches for an executable file called /init on the root of the initramfs image. If it finds it, it executes it as the init process with PID (process ID) set to 1. If it does not find it, it skips initramfs and proceeds with normal root file system processing. This logic is found in .../init/main.c. A character pointer called ramdisk_execute_command contains a pointer to this initialization command. By default it is set to the string "/init".

A kernel command-line parameter called rdinit=, when set, overrides this init specifier much the same way that init= does. To use it, simply add it to your kernel command line. For example, we could have set rdinit=/bin/sh on our kernel command line to directly call the busybox shell applet.

6.6 Shutdown

Orderly shutdown of an embedded system is often overlooked in a design. Improper shutdown can affect startup times and can even corrupt certain file system types. One of the more common complaints about using the EXT2 file system (the default in many desktop Linux distributions for several years) is the time it takes for an `fsck` (file system check) on startup after unplanned power loss. Servers with large disk systems can take many hours to properly `fsck` through a collection of large EXT2 partitions.

Each embedded project will likely have its own shutdown strategy. What works for one might or might not work for another. The scale of shutdown can range from a full System V shutdown scheme, to a simple script, to halt or reboot. Several Linux utilities are available to assist in the shutdown process, including the `shutdown`, `halt`, and `reboot` commands. Of course, these must be available for your chosen architecture.

A shutdown script should terminate all user space processes, which results in closing any open files used by those processes. If `init` is being used, issuing the command `init 0` halts the system. In general, the shutdown process first sends all processes the `SIGTERM` signal to notify them that the system is shutting down. A short delay ensures that all processes have the opportunity to perform their shutdown actions, such as closing files, saving state, and so on. Then all processes are sent the `SIGKILL` signal, which results in their termination. The shutdown process should attempt to unmount any mounted file systems and call the architecture-specific halt or reboot routines. The Linux `shutdown` command in conjunction with `init` exhibits this behavior.

6.7 Summary

This chapter presented an in-depth overview of user space initialization on a Linux kernel system. With this knowledge, you should be able to customize your own embedded system startup behavior.

- A root file system is required for all Linux systems. They can be difficult to build from scratch because of complex dependencies by each application.
- The File System Hierarchy standard provides guidance to developers for laying out a file system for maximum compatibility and flexibility.
- We presented a minimal file system as an example of how root file systems are created.

- The Linux kernel's final boot steps define and control a Linux system's startup behavior. Several mechanisms are available, depending on your embedded Linux system's requirements.
- The `init` process is a powerful system configuration and control utility that can serve as the basis for your own embedded Linux system. System initialization based on `init` was presented, along with sample startup script configurations.
- Initial ramdisk (`initrd`) is a Linux kernel feature to allow further startup behavior customization before mounting a final root file system and spawning `init`. We presented the mechanism and a sample configuration for using this powerful feature.
- `initramfs` simplifies the initial ramdisk mechanism while providing similar early startup facilities. It is easier to use, does not require loading a separate image, and is built automatically during each kernel build.

6.7.1 Suggestions for Additional Reading

File System Hierarchy Standard
Maintained by freestandards.org
www.pathname.com/fhs/

Boot process, init, and shutdown
Linux Documentation Project
http://tldp.org/LDP/intro-linux/html/sect_04_02.html

Init man page
Linux Documentation Project
http://tldp.org/LDP/sag/html/init-intro.html

A brief description of System V `init`
http://docs.kde.org/en/3.3/kdeadmin/ksysv/what-is-sysv-init.html

"Booting Linux: The History and the Future"
Werner Almesberger
www.almesberger.net/cv/papers/ols2k-9.ps

This page intentionally left blank