


Prentice Hall Open Source Software Development Series

Embedded Linux[®] Primer



A practical, real-world approach

CHRISTOPHER HALLINAN

Chapter 4. The Linux Kernel—A Different Perspective..... 1

Section 4.1. Background.....2

Section 4.2. Linux Kernel Construction..... 6

Section 4.3. Kernel Build System..... 15

Section 4.4. Obtaining a Linux Kernel..... 32

Section 4.5. Chapter Summary..... 33

Chapter 4



The Linux Kernel—A Different Perspective

In this chapter

- Background page 66
- Linux Kernel Construction page 70
- Kernel Build System page 79
- Obtaining a Linux Kernel page 96
- Chapter Summary page 97

If you want to learn about kernel internals, many good books are available on kernel design and operation. Several are presented in Section 4.5.1, “Suggestions for Additional Reading,” in this and other chapters throughout the book. However, very little has been written about how the kernel is organized and structured from a project perspective. What if you’re looking for the right place to add some custom support for your new embedded project? How do you know which files are important for your architecture?

At first glance, it might seem an almost impossible task to understand the Linux kernel and how to configure it for a specific platform or application. In a recent Linux kernel snapshot, the Linux kernel source tree consists of more than 20,000 files that contain more than six million lines—and that’s just the beginning. You still need tools, a root file system, and many Linux applications to make a usable system.

This chapter introduces the Linux kernel and covers how the kernel is organized and how the source tree is structured. We then examine the components that make up the kernel image and discuss the kernel source tree layout. Following this, we present the details of the kernel build system and the files that drive the kernel configuration and build system. This chapter concludes by examining what is required for a complete embedded Linux system.

4.1 Background

Linus Torvalds wrote the original version of Linux while he was a student at the University of Helsinki in Finland. His work began in 1991. In August of that year, Linus posted this now-famous announcement on `comp.os.minix`:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki
```

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat(same physical layout of the file-system (due to practical reasons)among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-).

Since that initial release, Linux has matured into a full-featured operating system with robustness, reliability, and high-end features that rival those of the best commercial operating systems. By some estimates, more than half of the Internet servers on the Web are powered by Linux servers. It is no secret that the online search giant Google uses a large collection of low-cost PCs running a fault-tolerant version of Linux to implement its popular search engine.

4.1.1 Kernel Versions

You can obtain the source code for a Linux kernel and complementary components in numerous places. Your local bookstore might have several versions as companion CD-ROMs in books about Linux. You also can download the kernel itself or even complete Linux distributions from numerous locations on the Internet. The official home for the Linux kernel is found at www.kernel.org. You will often hear the terms *mainline source* or *mainline kernel* referring to the source trees found at kernel.org.

As this book is being written, Linux Version 2.6 is the current version. Early in the development cycle, the developers chose a numbering system designed to differentiate between kernel source trees intended for development and experimentation and source trees intended to be stable, production-ready kernels. The numbering scheme contains a major version number, a minor version number, and then a sequence number. Before Linux Version 2.6, if the minor version number is even, it denotes a production kernel; if it is odd, it denotes a development kernel. For example:

- **Linux 2.4.x**—Production kernel
- **Linux 2.5.x**—Experimental (development)
- **Linux 2.6.x**—Production kernel

Currently, there is no separate development branch of the Linux 2.6 kernel. All new features, enhancements, and bug fixes are funneled through a series of gatekeepers who ultimately filter and push changes up to the top-level Linux source trees maintained by Andrew Morton and Linus Torvalds.

It is easy to tell what kernel version you are working with. The first few lines of the top-level *makefile*¹ in a kernel source tree detail the exact kernel version represented by a given instance. It looks like this for the 2.6.14 production kernel:

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 14
EXTRAVERSION =
NAME=Affluent Albatross
```

Later in the same makefile, these macros are used to form a version-level macro, like this:

```
KERNELRELEASE=$(VERSION) . $(PATCHLEVEL) . $(SUBLEVEL) $(EXTRAVERSION)
```

This macro is used in several places in the kernel source tree to indicate the kernel version. In fact, version information is used with sufficient frequency that the kernel developers have dedicated a set of macros derived from the version macros in the makefile. These macros are found in `.../include/linux/version.h`² in the Linux kernel source tree. They are reproduced here as Listing 4-1.

Listing 4-1

Kernel include File: `.../include/linux/version.h`

```
#define UTS_RELEASE "2.6.14"
#define LINUX_VERSION_CODE 132622
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
```

You can check the kernel version from a command prompt on a running Linux system like this:

```
$ cat /proc/version
Linux version 2.6.13 (chris@pluto) (gcc version 4.0.0 (DENX ELDK 4.0 4.0.0)) #2
Thu Feb 16 19:30:13 EST 2006
```

One final note about kernel versions: You can make it easy to keep track of the kernel version in your own kernel project by customizing the `EXTRAVERSION` field.

¹ We talk about the kernel build system and makefiles shortly.

² Throughout this book, three dots preceding any path are used to indicate whatever path it might take on your system to reach the top-level Linux source tree.

For example, if you were developing enhancements for some new kernel feature, you might set `EXTRAVERSION` to something like this:

```
EXTRAVERSION=-foo
```

Later, when you use `cat /proc/version`, you would see `Linux version 2.6.13-foo`, and this would help you distinguish between development versions of your own kernel.

4.1.2 Kernel Source Repositories

The official home for the kernel source code is www.kernel.org. There you can find both current and historical versions of the Linux kernel, as well as numerous patches. The primary FTP repository found at ftp.kernel.org contains subdirectories going all the way back to Linux Version 1.0. This site is the primary focus for the ongoing development activities within the Linux kernel.

If you download a recent Linux kernel from kernel.org, you will find files in the source tree for 25 different architectures and subarchitectures. Several other development trees support the major architectures. One of the reasons is simply the sheer volume of developers and changes to the kernel. If every developer on every architecture submitted patches to kernel.org, the maintainers would be inundated with changes and patch management, and would never get to do any feature development. As anyone involved with kernel development will tell you, it's already very busy!

Several other public source trees exist outside the mainline kernel.org source, mostly for architecture-specific development. For example, a developer working on the MIPS architecture might find a suitable kernel at www.linux-mips.org. Normally, work done in an architecture tree is eventually submitted to the kernel.org kernel. Most architecture developers try to sync up to the mainline kernel often, to keep up with new developments whenever possible. However, it is not always straightforward to get one's patches included in the mainline kernel, and there will always be a lag. Indeed, differences in the architecture kernel trees exist at any given point in time.

If you are wondering how to find a kernel for your particular application, the best way to proceed is to obtain the latest stable Linux source tree. Check to see if support for your particular processor exists, and then search the Linux kernel mailing lists for any patches or issues related to your application. Also find the mailing list that most closely matches your interest, and search that archive also.

Appendix E, “Open Source Resources,” contains several good references and sources of information related to kernel source repositories, mailing lists, and more.

4.2 Linux Kernel Construction

In the next few sections, we explore the layout, organization, and construction of the Linux kernel. Armed with this knowledge, you will find it much easier to navigate this large, complex source code base. Over time, there have been significant improvements in the organization of the source tree, especially in the architecture branch, which contains support for numerous architectures and specific machines. As this book is being written, an effort is underway to collapse the `ppc` and `ppc64` architecture branches into a single common `powerpc` branch. When the dust settles, there will be many improvements, including elimination of duplicate code, better organization of files, and partitioning of functionality.

4.2.1 Top-Level Source Directory

We make frequent reference to the *top-level source directory* throughout the book. In every case, we are referring to the highest-level directory contained in the kernel source tree. On any given machine, it might be located anywhere, but on a desktop Linux workstation, it is often found in `/usr/src/linux-x.y.z`, where `x.y.z` represents the kernel version. Throughout the book, we use the shorthand `.../` to represent the top-level kernel source directory.

The top-level kernel source directory contains the following subdirectories. (We have omitted the nondirectory entries in this listing, as well as directories used for source control for clarity and brevity.)

arch	crypto	Documentation	drivers	fs	include
init	ipc	kernel	lib	mm	net
scripts	security	sound	usr		

Many of these subdirectories contain several additional levels of subdirectories containing source code, makefiles, and configuration files. By far the largest branch of the Linux kernel source tree is found under `.../drivers`. Here you can find support for Ethernet network cards, USB controllers, and the numerous hardware devices that the Linux kernel supports. As you might imagine, the `.../arch` subdirectory is the next largest, containing support for more than 20 unique processor architectures.

Additional files found in the top-level Linux subdirectory include the top-level makefile, a hidden configuration file (*dot-config*, introduced in Section 4.3.1, “The Dot-Config”) and various other informational files not involved in the build itself. Finally, two important build targets are found in the top-level kernel source tree after a successful build: `System.map` and the kernel proper, `vmlinux`. Both are described shortly.

4.2.2 Compiling the Kernel

Understanding a large body of software such as Linux can be a daunting task. It is too large to simply “step through” the code to follow what is happening. Multithreading and preemption add to the complexity of analysis. In fact, even locating the entry point (the first line of code to be executed upon entry to the kernel) can be challenging. One of the more useful ways to understand the structure of a large binary image is to examine its built components.

The output of the kernel build system produces several common files, as well as one or more architecture-specific binary modules. Common files are always built regardless of the architecture. Two of the common files are `System.map` and `vmlinux`, introduced earlier. The former is useful during kernel debug and is particularly interesting. It contains a human-readable list of the kernel symbols and their respective addresses. The latter is an architecture-specific *ELF*³ file in executable format. It is produced by the top-level kernel makefile for every architecture. If the kernel was compiled with symbolic debug information, it will be contained in the `vmlinux` image. In practice, although it is an ELF executable, this file is usually *never* booted directly, as you will see shortly.

Listing 4-2 is a snippet of output resulting from executing `make` in a recent kernel tree configured for the ARM XScale architecture. The kernel source tree was configured for the ADI Engineering Coyote reference board based on the Intel IXP425 network processor using the following command:

```
make ARCH=arm CROSS_COMPILE=xscale_be- ixp4xx_defconfig
```

This command does not build the kernel; it prepares the kernel source tree for the XScale architecture including an initial default configuration for this architecture and processor. It builds a default configuration (the *dot-config* file) that drives

³ Executable and Linking Format, a de-facto standard format for binary executable files.

the kernel build, based on the defaults found in the `ixp4xx_defconfig` file. We have more to say about the configuration process later, in Section 4.3, “Kernel Build System.”

Listing 4-2 shows the command that builds the kernel. Only the first few and last few lines of the build output are shown for this discussion.

Listing 4-2

Kernel Build Output

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- zImage
CHK      include/linux/version.h
HOSTCC   scripts/basic/fixdep
.
. <hundreds of lines of output omitted here>
.
LD       vmlinux
SYSMAP   System.map
SYSMAP   .tmp_System.map
OBJCOPY  arch/arm/boot/Image
Kernel:  arch/arm/boot/Image is ready
AS       arch/arm/boot/compressed/head.o
GZIP     arch/arm/boot/compressed/piggy.gz
AS       arch/arm/boot/compressed/piggy.o
CC       arch/arm/boot/compressed/misc.o
AS       arch/arm/boot/compressed/head-xscale.o
AS       arch/arm/boot/compressed/big-endian.o
LD       arch/arm/boot/compressed/vmlinux
OBJCOPY  arch/arm/boot/zImage
Kernel:  arch/arm/boot/zImage is ready
Building modules, stage 2.
...
```

To begin, notice the invocation of the build. Both the desired architecture (`ARCH=arm`) and the toolchain (`CROSS_COMPILE=xscale_be-`) were specified on the command line. This forces `make` to use the XScale toolchain to build the kernel image and to use the arm-specific branch of the kernel source tree for architecture-dependent portions of the build. We also specify a target called `zImage`. This target is common to many architectures and is described in Chapter 5, “Kernel Initialization.”

The next thing you might notice is that the actual commands used for each step have been hidden and replaced with a shorthand notation. The motivation behind this was to clean up the build output to draw more attention to intermediate build

issues, particularly compiler warnings. In earlier kernel source trees, each compilation or link command was output to the console verbosely, which often required several lines for each step. The end result was virtually unreadable, and compiler warnings slipped by unnoticed in the noise. The new system is definitely an improvement because any anomaly in the build process is easily spotted. If you want or need to see the complete build step, you can force verbose output by defining `V=1` on the `make` command line.

We have omitted most of the actual compilation and link steps in Listing 4-2, for clarity. (This particular build has more than 900 individual compile and link commands in the build. That would have made for a long listing, indeed.) After all the intermediate files and library archives have been built and compiled, they are put together in one large ELF build target called `vmlinux`. Although it is architecture specific, this `vmlinux` target is a common target—it is produced for all supported Linux architectures.

4.2.3 The Kernel Proper: `vmlinux`

Notice this line in Listing 4-2:

```
LD /arch/arm/boot/compressed/vmlinux
```

The `vmlinux` file is the actual *kernel proper*. It is a fully stand-alone, monolithic image. No unresolved external references exist within the `vmlinux` binary. When caused to execute in the proper context (by a bootloader designed to boot the Linux kernel), it boots the board on which it is running, leaving a completely functional kernel.

In keeping with the philosophy that to understand a system one must first understand its parts, let's look at the construction of the `vmlinux` kernel object.

Listing 4-3 reproduces the actual link stage of the build process that resulted in the `vmlinux` ELF object. We have formatted it with line breaks (indicated by the UNIX line-continuation character, ‘\’) to make it more readable, but otherwise it is the exact output produced by the `vmlinux` link step in the build process from Listing 4-2. If you were building the kernel by hand, this is the link command you would issue from the command line.

Listing 4-3

Link Stage: `vmlinux`

```

xscale_be-ld -EB -p --no-undefined -X -o vmlinux \
-T arch/arm/kernel/vmlinux.lds \
arch/arm/kernel/head.o \
arch/arm/kernel/init_task.o \
init/built-in.o \
--start-group \
usr/built-in.o \
arch/arm/kernel/built-in.o \
arch/arm/mm/built-in.o \
arch/arm/common/built-in.o \
arch/arm/mach-ixp4xx/built-in.o \
arch/arm/nwfpe/built-in.o \
kernel/built-in.o \
mm/built-in.o \
fs/built-in.o \
ipc/built-in.o \
security/built-in.o \
crypto/built-in.o \
lib/lib.a \
arch/arm/lib/lib.a \
lib/built-in.o \
arch/arm/lib/built-in.o \
drivers/built-in.o \
sound/built-in.o \
net/built-in.o \
--end-group \
.tmp_kallsyms2.o

```

4.2.4 Kernel Image Components

From Listing 4-3, you can see that the `vmlinux` image consists of several composite binary images. Right now, it is not important to understand the purpose of each component. What is important is to understand the top-level view of what components make up the kernel. The first line of the link command in Listing 4-3 specifies the output file (`-o vmlinux.`) The second line specifies the *linker script* file (`-T vmlinux.lds`), a detailed recipe for how the kernel binary image should be linked.⁴

The third and subsequent lines from Listing 4-3 specify the object modules that form the resulting binary image. Notice that the first object specified is `head.o`. This object was assembled from `/arch/arm/kernel/head.S`, an architecture-specific assembly language source file that performs very low-level kernel initialization. If you were searching for the first line of code to be executed by the kernel, it would make sense to start your search here because it will ultimately be the first code found in the binary image created by this link stage. We examine kernel initialization in detail in Chapter 5.

The next object, `init_task.o`, sets up initial thread and task structures that the kernel requires. Following this is a large collection of object modules, each having a common name: `built-in.o`. You will notice, however, that each `built-in.o` object comes from a specific part of the kernel source tree, as indicated by the path component preceding the `built-in.o` object name. These are the binary objects that are included in the kernel image. An illustration might make this clearer.

⁴ The linker script file has a peculiar syntax. The details can be found in the documentation for the GNU linker.

Figure 4-1 illustrates the binary makeup of the `vmlinux` image. It contains a section for each line of the link stage. It's not to scale because of space considerations, but you can see the relative sizes of each functional component.

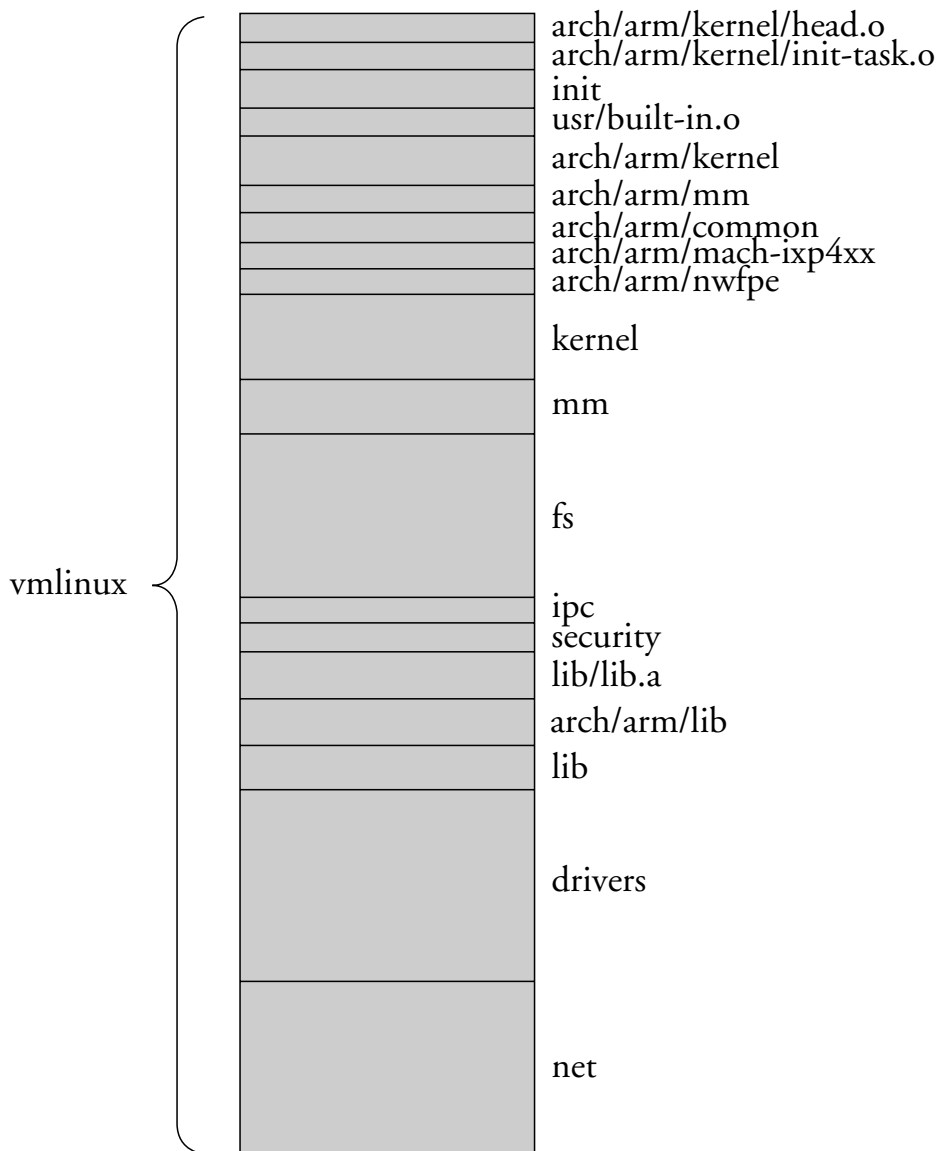


FIGURE 4-1
vmlinux image components

It might come as no surprise that the three largest binary components are the file system code, the network code, and all the built-in drivers. If you take the kernel code and the architecture-specific kernel code together, this is the next-largest binary component. Here you find the scheduler, process and thread management, timer management, and other core kernel functionality. Naturally, the kernel contains some architecture-specific functionality, such as low-level context switching, hardware-level interrupt and timer processing, processor exception handling, and more. This is found in `.../arch/arm/kernel`.

Bear in mind that we are looking at a specific example of a kernel build. In this particular example, we are building a kernel specific to the ARM XScale architecture and, more specifically, the Intel IXP425 network processor on the ADI Engineering reference board. You can see the machine-specific binary components in Figure 4-1 as `arch/arm/mach-ixp4xx`. Each architecture and machine type (processor/reference board) has different elements in the architecture-specific portions of the kernel, so the makeup of the `vmlinux` image is slightly different. When you understand one example, you will find it easy to navigate others.

To help you understand the breakdown of functionality in the kernel source tree, Table 4-1 lists each component in Figure 4-1, together with a short description of each binary element that makes up the `vmlinux` image.

TABLE 4-1
vmlinux Image Components Description

Component	Description
<code>arch/arm/kernel/head.o</code>	Kernel architecture-specific startup code.
<code>init_task.o</code>	Initial thread and task structs required by kernel.
<code>init/built-in.o</code>	Main kernel-initialization code. See Chapter 5.
<code>usr/built-in.o</code>	Built-in <code>initramfs</code> image. See Chapter 5.
<code>arch/arm/kernel/built-in.o</code>	Architecture-specific kernel code.
<code>arch/arm/mm/built-in.o</code>	Architecture-specific memory-management code.
<code>arch/arm/common/built-in.o</code>	Architecture-specific generic code. Varies by architecture.

(continues)

TABLE 4-1 (Continued)

Component	Description
arch/arm/mach-ixp4xx/built-in.o	Machine-specific code, usually initialization.
arch/arm/nwfpe/built-in.o	Architecture-specific floating point-emulation code.
kernel/built-in.o	Common components of the kernel itself.
mm/built-in.o	Common components of memory-management code.
ipc/built-in.o	Interprocess communications, such as SysV IPC.
security/built-in.o	Linux security components.
lib/lib.a	Archive of miscellaneous helper functions.
arch/arm/lib/lib.a	Architecture-specific common facilities. Varies by architecture.
lib/built-in.o	Common kernel helper functions.
drivers/built-in.o	All the built-in drivers—not loadable modules.
sound/built-in.o	Sound drivers.
net/built-in.o	Linux networking.
.tmp_kallsyms2.o	Symbol table.

When we speak of the kernel proper, this `vmlinux` image is being referenced. As mentioned earlier, very few platforms boot this image directly. For one thing, it is almost universally compressed. At a bare minimum, a bootloader must decompress the image. Many platforms require some type of stub bolted onto the image to perform the decompression. Later in Chapter 5, you will learn how this image is packaged for different architectures, machine types, and bootloaders, and the requirements for booting it.

4.2.5 Subdirectory Layout

Now that you've seen how the build system controls the kernel image, let's take a look at a representative kernel subdirectory. Listing 4-4 details the contents of the `mach-ixp425` subdirectory. This directory exists under the `.../arch/arm` architecture-specific branch of the source tree.

Listing 4-4

Kernel Subdirectory

```
$ ls -l linux-2.6/arch/arm/mach-ixp425
total 92
-rw-rw-r-- 1 chris chris 11892 Oct 10 14:53 built-in.o
-rw-rw-r-- 1 chris chris 6924 Sep 29 15:39 common.c
-rw-rw-r-- 1 chris chris 3525 Oct 10 14:53 common.o
-rw-rw-r-- 1 chris chris 13062 Sep 29 15:39 common-pci.c
-rw-rw-r-- 1 chris chris 7504 Oct 10 14:53 common-pci.o
-rw-rw-r-- 1 chris chris 1728 Sep 29 15:39 coyote-pci.c
-rw-rw-r-- 1 chris chris 1572 Oct 10 14:53 coyote-pci.o
-rw-rw-r-- 1 chris chris 2118 Sep 29 15:39 coyote-setup.c
-rw-rw-r-- 1 chris chris 2180 Oct 10 14:53 coyote-setup.o
-rw-rw-r-- 1 chris chris 2042 Sep 29 15:39 ixdp425-pci.c
-rw-rw-r-- 1 chris chris 3656 Sep 29 15:39 ixdp425-setup.c
-rw-rw-r-- 1 chris chris 2761 Sep 29 15:39 Kconfig
-rw-rw-r-- 1 chris chris 259 Sep 29 15:39 Makefile
-rw-rw-r-- 1 chris chris 3102 Sep 29 15:39 prpmc1100-pci.c
```

The directory contents in Listing 4-4 have common components found in many kernel source subdirectories: `Makefile` and `Kconfig`. These two files drive the kernel configuration-and-build process. Let's look at how that works.

4.3 Kernel Build System

The Linux kernel configuration and build system is rather complicated, as one would expect of software projects containing more than six million lines of code! In this section, we cover the foundation of the kernel build system for developers who need to customize the build environment.

A recent Linux kernel snapshot showed more than 800 makefiles⁵ in the kernel source tree. This might sound like a large number, but it might not seem so large

⁵ Not all these makefiles are directly involved in building the kernel. Some, for example, build documentation files.

when you understand the structure and operation of the build system. The Linux kernel build system has been significantly updated since the days of Linux 2.4 and earlier. For those of you familiar with the older kernel build system, we're sure you will find the new *Kbuild* system to be a huge improvement. We limit our discussion in this section to this and later kernel versions based on *Kbuild*.

4.3.1 The Dot-Config

Introduced earlier, the dot-config file is the configuration blueprint for building a Linux kernel image. You will likely spend significant effort at the start of your Linux project building a configuration that is appropriate for your embedded platform. Several editors, both text based and graphical, are designed to edit your kernel configuration. The output of this configuration exercise is written to a configuration file named `.config`, located in the top-level Linux source directory that drives the kernel build.

You have likely invested significant time perfecting your kernel configuration, so you will want to protect it. Several `make` commands delete this configuration file without warning. The most common is `make mrproper`. This `make` target is designed to return the kernel source tree to its pristine, unconfigured state. This includes removing all configuration data from the source tree—and, yes, it deletes your `.config`.

As you might know, any filename in Linux preceded by a dot is a hidden file in Linux. It is unfortunate that such an important file is marked hidden; this has brought considerable grief to more than one developer. If you execute `make mrproper` without having a backup copy of your `.config` file, you, too, will share our grief. (You have been warned—back up your `.config` file!)

The `.config` file is a collection of definitions with a simple format. Listing 4.5 shows a snippet of a `.config` from a recent Linux kernel release.

Listing 4-5

Snippet from Linux 2.6 `.config`

```
...
# USB support
#
CONFIG_USB=m
# CONFIG_USB_DEBUG is not set

# Miscellaneous USB options
#
CONFIG_USB_DEVICEFS=y
# CONFIG_USB_BANDWIDTH is not set
# CONFIG_USB_DYNAMIC_MINORS is not set

# USB Host Controller Drivers
#
CONFIG_USB_EHCI_HCD=m
# CONFIG_USB_EHCI_SPLIT_ISO is not set
# CONFIG_USB_EHCI_ROOT_HUB_TT is not set
CONFIG_USB_OHCI_HCD=m
CONFIG_USB_UHCI_HCD=m
...
```

To understand the `.config` file, you need to understand a fundamental aspect of the Linux kernel. Linux has a monolithic structure. That is, the entire kernel is compiled and linked as a single statically linked executable. However, it is possible to compile and *incrementally link*⁶ a set of sources into a single object module suitable for dynamic insertion into a running kernel. This is the usual method for supporting most common device drivers. In Linux, these are called *loadable modules*. They are also generically called device drivers. After the kernel is booted, a special application program is invoked to insert the loadable module into a running kernel.

Armed with that knowledge, let's look again at Listing 4-5. This snippet of the configuration file (`.config`) shows a portion of the USB subsystem configuration. The first configuration option, `CONFIG_USB=m`, declares that the USB subsystem is to be included in this kernel configuration and that it will be compiled as a *dynamically loadable module* (`=m`), to be loaded sometime after the kernel has booted. The other

⁶ Incremental linking is a technique used to generate an object module that is intended to be linked again into another object. In this way, unresolved symbols that remain after incremental linking do not generate errors—they are resolved at the next link stage.

choice would have been `=y`, in which case the USB module would be compiled and statically linked as part of the kernel image itself. It would end up in the `.../drivers/built-in.o` composite binary that you saw in Listing 4-3 and Figure 4-1. The astute reader will realize that if a driver is configured as a loadable module, its code is not included in the kernel proper, but rather exists as a stand-alone object module, a *loadable module*, to be inserted into the running kernel after boot.

Notice in Listing 4-5 the `CONFIG_USB_DEVICEFS=y` declaration. This configuration option behaves in a slightly different manner. In this case, `USB_DEVICEFS` (as configuration options are commonly abbreviated) is not a stand-alone module, but rather a feature to be enabled or disabled in the USB driver. It does not necessarily result in a module that is compiled into the kernel proper (`=y`); instead, it enables one or more features, often represented as additional object modules to be included in the overall USB device driver module. Usually, the help text in the configuration editor, or the hierarchy presented by the configuration editor, makes this distinction clear.

4.3.2 Configuration Editor(s)

Early kernels used a simple command line driven script to configure the kernel. This was cumbersome even for early kernels, in which the number of configuration parameters was much smaller. This command line style interface is still supported, but using it is tedious, to say the least. A typical configuration from a recent kernel requires answering more than 600 questions from the command line, entering your choice followed by the Enter key for each query from the script. Furthermore, if you make a mistake, there is no way to back up; you must start from the beginning again. That can be profoundly frustrating if you make a mistake on the 599th entry!

In some situations, such as building a kernel on an embedded system without graphics, using the command line configuration utility is unavoidable, but this author would go to great lengths to find a way around it.

The kernel-configuration subsystem uses several graphical front ends. In fact, a recent Linux kernel release included 10 such configuration targets. They are summarized here, from text taken directly from the output of `make help`:

- `config`—Update current config using a line-oriented program
- `menuconfig`—Update current config using a menu-based program
- `xconfig`—Update current config using a QT-based front end
- `gconfig`—Update current config using a GTK-based front end

- `oldconfig`—Update current config using a provided `.config` as the base
- `randconfig`—New config with random answer to all options
- `defconfig`—New config with default answer to all options
- `allmodconfig`—New config that selects modules, when possible
- `allyesconfig`—New config in which all options are accepted with `yes`
- `allnoconfig`—New minimal config

The first four of these makefile configuration targets invoke a form of configuration editor, as described in the list. Because of space considerations, we focus our discussion in this chapter and others only on the GTK-based graphical front end. Realize that you can use the configuration editor of your choice with the same results.

The configuration editor is invoked by entering the command `make gconfig` from the top-level kernel directory.⁷ Figure 4-2 shows the top-level configuration menu presented to the developer when `gconfig` is run. From here, every available configuration parameter can be accessed to generate a custom kernel configuration.

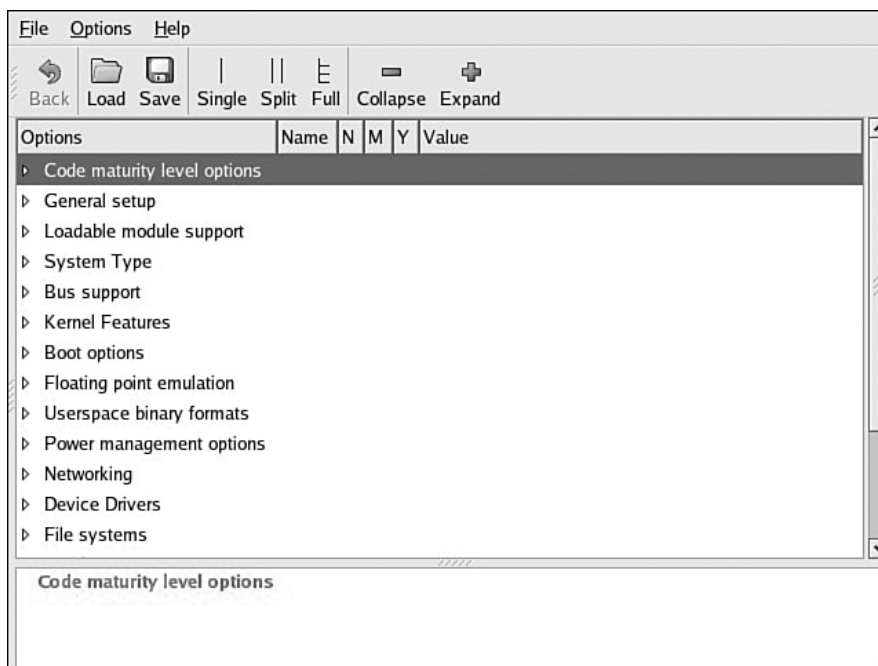


FIGURE 4-2
Top-level kernel configuration

⁷ As mentioned, you can use the configuration editor of your choice, such as `make xconfig` or `make menuconfig`.

When the configuration editor is exited, you are prompted to save your changes. If you elect to save your changes, the global configuration file `.config` is updated (or created, if it does not already exist). This `.config` file, introduced earlier, drives the kernel build via the top-level makefile. You will notice in this makefile that the `.config` file is read directly by an `include` statement.

Most kernel software modules also read the configuration indirectly via the `.config` file as follows. During the build process, the `.config` file is processed into a C header file found in the `.../include/linux` directory, called `autoconf.h`. This is an automatically generated file and should never be edited directly because edits are lost each time a configuration editor is run. Many kernel source files include this file directly using the `#include` preprocessor directive. Listing 4-6 reproduces a section of this header file that corresponds to the earlier USB example above. Note that, for each entry in the `.config` file snippet in Listing 4-5, a corresponding entry is created in `autoconf.h`. This is how the source files in the kernel source tree reference the kernel configuration.

Listing 4-6

Linux `autoconf.h`

```
/*
 * USB support
 */
#define CONFIG_USB_MODULE 1
#undef CONFIG_USB_DEBUG

/*
 * Miscellaneous USB options
 */
#define CONFIG_USB_DEVICEFS 1
#undef CONFIG_USB_BANDWIDTH
#undef CONFIG_USB_DYNAMIC_MINORS

/*
 * USB Host Controller Drivers
 */
#define CONFIG_USB_EHCI_HCD_MODULE 1
#undef CONFIG_USB_EHCI_SPLIT_ISO
#undef CONFIG_USB_EHCI_ROOT_HUB_TT
#define CONFIG_USB_OHCI_HCD_MODULE 1
#define CONFIG_USB_UHCI_HCD_MODULE 1
```

If you haven't already done so, execute `make gconfig` in your top-level kernel source directory, and poke around this configuration utility to see the large number of subsections and configuration options available to the Linux developer. As long as you don't explicitly save your changes, they are lost upon exiting the configuration editor and you can safely explore without modifying your kernel configuration.⁸ Many configuration parameters contain helpful explanation text, which can add to your understanding of the different configuration options.

4.3.3 Makefile Targets

If you type `make help` at the top-level Linux source directory, you are presented with a list of targets that can be generated from the source tree. The most common use of `make` is to specify no target. This generates the kernel ELF file `vmlinux` and is the default binary image for your chosen architecture (for example, `bzImage` for x86). Specifying `make` with no target also builds all the device-driver modules (kernel-loadable modules) specified by the configuration.

Many architectures and machine types require binary targets specific to the architecture and bootloader in use. One of the more common architecture-specific targets is `zImage`. In many architectures, this is the default target image that can be loaded and run on the target embedded system. One of the common mistakes that newcomers make is to specify `bzImage` as the `make` target. The `bzImage` target is specific to the x86/PC architecture. Contrary to popular myth, the `bzImage` is not a `bzip2`-compressed image. It is a big `zImage`. Without going into the details of legacy PC architecture, it is enough to know that a `bzImage` is suitable only for PC-compatible machines with an industry-standard PC-style BIOS.

⁸ Better yet, make a backup copy of your `.config` file.

Listing 4-7 contains the output from `make help` from a recent Linux kernel. You can see from the listing that many targets are available in the top-level Linux kernel makefile. Each is listed along with a short description of its use. It is important to realize that even the `help make` target (as in `make help`) is architecture specific. You get a different list of architecture-specific targets depending on the architecture you pass on the `make` invocation. Listing 4-7 illustrates an invocation that specifies the ARM architecture, as you can see from the `make` command line.

Listing 4-7

Makefile Targets

```
$ make ARCH=arm help
Cleaning targets:
  clean          - remove most generated files but keep the config
  mrproper       - remove all generated files + config + various backup files

Configuration targets:
  config         - Update current config utilising a line-oriented program
  menuconfig     - Update current config utilising a menu based program
  xconfig        - Update current config utilising a QT based front-end
  gconfig        - Update current config utilising a GTK based front-end
  oldconfig      - Update current config utilising a provided .config as base
  randconfig     - New config with random answer to all options
  defconfig      - New config with default answer to all options
  allmodconfig   - New config selecting modules when possible
  allyesconfig   - New config where all options are accepted with yes
  allnoconfig    - New minimal config

Other generic targets:
  all            - Build all targets marked with [*]
* vmlinux       - Build the bare kernel
* modules       - Build all modules
  modules_install - Install all modules
  dir/          - Build all files in dir and below
  dir/file.[ois] - Build specified target only
  dir/file.ko    - Build module including final link
  rpm           - Build a kernel as an RPM package
  tags/TAGS     - Generate tags file for editors
  cscope        - Generate cscope index
  kernelrelease - Output the release version string

Static analysers
  buildcheck    - List dangling references to vmlinux discarded sections and
                  init sections from non-init sections
  checkstack    - Generate a list of stack hogs
  namespacecheck - Name space analysis on compiled kernel

Kernel packaging:
  rpm-pkg       - Build the kernel as an RPM package
  binrpm-pkg    - Build an rpm package containing the compiled kernel and
                  modules
```



```

deb-pkg      - Build the kernel as an deb package
tar-pkg      - Build the kernel as an uncompressed tarball
targz-pkg    - Build the kernel as a gzip compressed tarball
tarbz2-pkg   - Build the kernel as a bzip2 compressed tarball

```

Documentation targets:

```

Linux kernel internal documentation in different formats:
xmldocs (XML DocBook), psdocs (Postscript), pdfdocs (PDF)
htmldocs (HTML), mandocs (man pages, use installmandocs to install)

```

Architecture specific targets (arm):

```

* zImage      - Compressed kernel image (arch/arm/boot/zImage)
Image         - Uncompressed kernel image (arch/arm/boot/Image)
* xipImage     - XIP kernel image, if configured (arch/arm/boot/xipImage)
bootpImage    - Combined zImage and initial RAM disk
                (supply initrd image via make variable INITRD=<path>)
install       - Install uncompressed kernel
zinstall      - Install compressed kernel
                Install using (your) ~/bin/installkernel or
                (distribution) /sbin/installkernel or
                install to $(INSTALL_PATH) and run lilo

```

```

assabet_defconfig      - Build for assabet
badge4_defconfig       - Build for badge4
bast_defconfig         - Build for bast
cerfcube_defconfig     - Build for cerfcube
clps7500_defconfig     - Build for clps7500
collie_defconfig       - Build for collie
corgi_defconfig        - Build for corgi
ebsa110_defconfig      - Build for ebsa110
edb7211_defconfig      - Build for edb7211
enp2611_defconfig      - Build for enp2611
ep80219_defconfig      - Build for ep80219
epxa10db_defconfig     - Build for epxa10db
footbridge_defconfig   - Build for footbridge
fortunet_defconfig     - Build for fortunet
h3600_defconfig        - Build for h3600
h7201_defconfig        - Build for h7201
h7202_defconfig        - Build for h7202
hackkit_defconfig      - Build for hackkit
integrator_defconfig   - Build for integrator
iq31244_defconfig      - Build for iq31244
iq80321_defconfig      - Build for iq80321
iq80331_defconfig      - Build for iq80331
iq80332_defconfig      - Build for iq80332
ixdp2400_defconfig     - Build for ixdp2400
ixdp2401_defconfig     - Build for ixdp2401
ixdp2800_defconfig     - Build for ixdp2800
ixdp2801_defconfig     - Build for ixdp2801
ixp4xx_defconfig       - Build for ixp4xx
jornada720_defconfig   - Build for jornada720
lart_defconfig         - Build for lart
lpd7a400_defconfig     - Build for lpd7a400
lpd7a404_defconfig     - Build for lpd7a404
lubbock_defconfig      - Build for lubbock

```

```

lusl7200_defconfig      - Build for lusl7200
mainstone_defconfig     - Build for mainstone
mx1ads_defconfig        - Build for mx1ads
neponset_defconfig      - Build for neponset
netwinder_defconfig     - Build for netwinder
omap_h2_1610_defconfig  - Build for omap_h2_1610
pleb_defconfig          - Build for pleb
poodle_defconfig        - Build for poodle
pxa255-idp_defconfig    - Build for pxa255-idp
rpc_defconfig           - Build for rpc
s3c2410_defconfig       - Build for s3c2410
shannon_defconfig       - Build for shannon
shark_defconfig         - Build for shark
simpad_defconfig        - Build for simpad
smdk2410_defconfig      - Build for smdk2410
spitz_defconfig         - Build for spitz
versatile_defconfig     - Build for versatile

make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
make O=dir [targets] Locate all output files in "dir", including .config
make C=1 [targets] Check all c source with $CHECK (sparse)
make C=2 [targets] Force check of all c source with $CHECK (sparse)

```

Execute "make" or "make all" to build all targets marked with [*]
 For further info see the ./README file

Many of these targets you might never use. However, it is useful to know that they exist. As you can see from Listing 4-7, the targets listed with an asterisk are built by default. Notice the numerous default configurations, listed as **_defconfig*. Recall from Section 4.2.2, “Compiling the Kernel,” the command we used to preconfigure a pristine kernel source tree: We invoked `make` with an architecture and a default configuration. The default configuration was *ixp4xx_defconfig*, which appears in this list of ARM targets. This is a good way to discover all the default configurations available for a particular kernel release and architecture.

4.3.4 Kernel Configuration

`Kconfig` (or a file with a similar root followed by an extension, such as `Kconfig.ext`) exists in almost 300 kernel subdirectories. `Kconfig` drives the configuration process for the features contained within its subdirectory. The contents of `Kconfig` are parsed by the configuration subsystem, which presents configuration choices to the user, and contains help text associated with a given configuration parameter.

The configuration utility (such as `gconf`, presented earlier) reads the `Kconfig` files starting from the arch subdirectory's `Kconfig` file. It is invoked from the

Kconfig makefile with an entry that looks like this:

```
gconf: $(obj)/gconf
    $< arch/$(ARCH)/Kconfig
```

Depending on which architecture you are building, `gconf` reads this architecture-specific `Kconfig` as the top-level configuration definition. Contained within `Kconfig` are a number of lines that look like this:

```
source "drivers/pci/Kconfig"
```

This directive tells the configuration editor utility to read in another `Kconfig` file from another location within the kernel source tree. Each architecture contains many such `Kconfig` files; taken together, these determine the complete set of menu options presented to the user when configuring the kernel. Each `Kconfig` file is free to source additional `Kconfig` files in different parts of the source tree. The configuration utility—`gconf`, in this case, recursively reads the `Kconfig` file chain and builds the configuration menu structure.

Listing 4-8 is a partial tree view of the `Kconfig` files associated with the ARM architecture. In a recent Linux 2.6 source tree from which this example was taken, the kernel configuration was defined by 170 separate `Kconfig` files. This listing omits most of those, for the sake of space and clarity—the idea is to show the overall structure. To list them all in this tree view would take several pages of this text.

Listing 4-8

Partial Listing of Kconfig for ARM Architecture

```
arch/arm/Kconfig <<<<<< (top level Kconfig)
|->  init/Kconfig
|   ...
|->  arch/arm/mach-iop3xx/Kconfig
|->  arch/arm/mach-ixp4xx/Kconfig
|   ...
|->  net/Kconfig
|   |-->  net/ipv4/Kconfig
|   |     |-->  net/ipv4/ipvs/Kconfig
|   |     ...
|->  drivers/char/Kconfig
|   |-->  drivers/serial/Kconfig
|   ...
|->  drivers/usb/Kconfig
|   |-->  drivers/usb/core/Kconfig
|   |-->  drivers/usb/host/Kconfig
|   ...
|->  lib/Kconfig
```

a separate version of the kernel source tree for each hardware version, a developer can add configuration options to enable his custom features.

The configuration management architecture described in the previous paragraphs makes it easy to customize and add features. A quick peek into a typical `Kconfig` file shows the structure of the configuration script language. As an example, assume that you have two hardware platforms based on the IXP425 network processor, and that your engineering team had dubbed them Vega and Constellation. Each board has specialized hardware that must be initialized early during the kernel boot phase. Let's see how easy it is to add these configuration options to the set of choices presented to the developer during kernel configuration. Listing 4-9 is a snippet from the top-level ARM `Kconfig` file.

Listing 4-9

Snippet from `.../arch/arm/Kconfig`

```
source "init/Kconfig"

menu "System Type"

choice
    prompt "ARM system type"
    default ARCH_RPC

config ARCH_CLPS7500
    bool "Cirrus-CL-PS7500FE"

config ARCH_CLPS711X
    bool "CLPS711x/EP721x-based"

...

source "arch/arm/mach-ixp4xx/Kconfig"
```

In this `Kconfig` snippet taken from the top-level ARM architecture `Kconfig`, you see the menu item `System Type` being defined. After the `ARM system type` prompt, you see a list of choices related to the ARM architecture. Later in the file, you see the inclusion of the IXP4xx-specific `Kconfig` definitions. In this file, you add your custom

configuration switches. Listing 4-10 reproduces a snippet of this file. Again, for readability and convenience, we've omitted irrelevant text, as indicated by the ellipsis.

Listing 4-10

File Snippet: `arch/arm/mach-ixp4xx/Kconfig`

```
menu "Intel IXP4xx Implementation Options"

comment "IXP4xx Platforms"

config ARCH_AVILA
    bool "Avila"
    help
        Say 'Y' here if you want your kernel to support...

config ARCH_ADI_COYOTE
    bool "Coyote"
    help
        Say 'Y' here if you want your kernel to support
        the ADI Engineering Coyote...

# (These are our new custom options)
config ARCH_VEGA
    bool "Vega"
    help
        Select this option for "Vega" hardware support

config ARCH_CONSTELLATION
    bool "Constellation"
    help
        Select this option for "Constellation"
        hardware support

...
```

Figure 4-4 illustrates the result of these changes as it appears when running the `gconf` utility (via `make ARCH=arm gconfig`). As a result of these simple changes, the configuration editor now includes options for our two new hardware platforms.⁹ Shortly, you'll see how you can use this configuration information in the source tree to conditionally select objects that contain support for your new boards.

⁹ We have intentionally removed many options under ARM system type and Intel IXP4xx Implementation Options to fit the picture on the page.

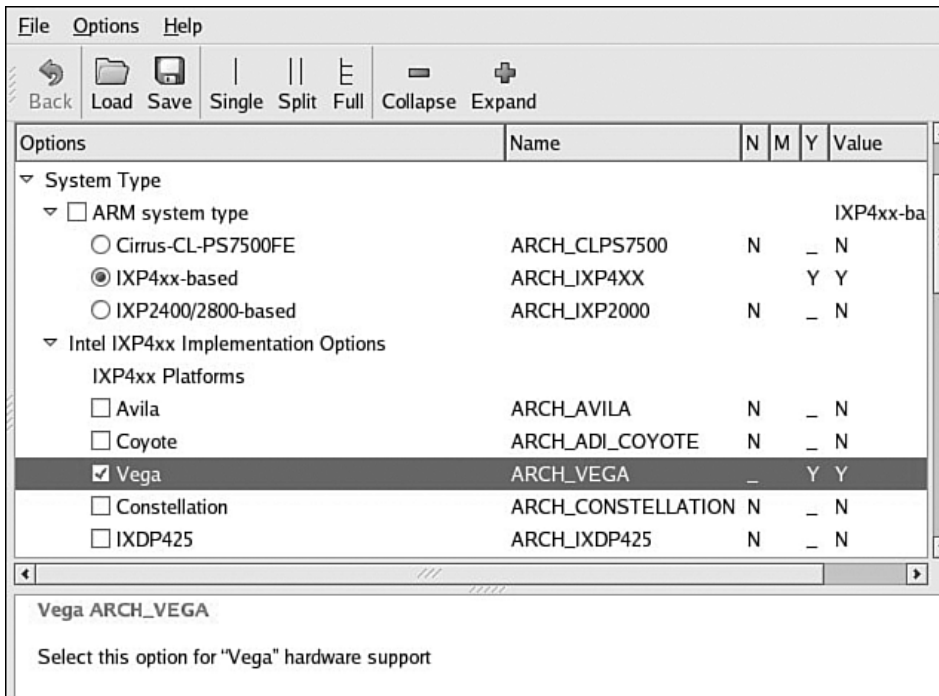


FIGURE 4-4
Custom configuration options

After the configuration editor (`gconf`, in these examples) is run and you select support for one of your custom hardware platforms, the `.config` file introduced earlier contains macros for your new options. As with all kernel-configuration options, each is preceded with `CONFIG_` to identify it as a kernel-configuration option. As a result, two new configuration options have been defined, and their state has been recorded in the `.config` file. Listing 4-11 shows the new `.config` file with your new configuration options.

Listing 4-11

Customized `.config` File Snippet

```
...
#
# IXP4xx Platforms
#
# CONFIG_ARCH_AVILA is not set
# CONFIG_ARCH_ADI_COYOTE is not set
CONFIG_ARCH_VEGA=y
# CONFIG_ARCH_CONSTELLATION is not set
# CONFIG_ARCH_IXDP425 is not set
# CONFIG_ARCH_PRPMC1100 is not set
...
```

Notice two new configuration options related to your Vega and Constellation hardware platforms. As illustrated in Figure 4-4, you selected support for Vega; in the `.config` file, you can see the new `CONFIG_` option representing that the Vega board is selected and set to the value `'y'`. Notice also that the `CONFIG_` option related to Constellation is present but not selected.

4.3.6 Kernel Makefiles

When building the kernel, the Makefiles scan the configuration and decide what subdirectories to descend into and what source files to compile for a given configuration. To complete the example of adding support for two custom hardware platforms, Vega and Constellation, let's look at the makefile that would read this configuration and take some action based on customizations.

Because you're dealing with hardware specific options in this example, assume that the customizations are represented by two hardware-setup modules called `vega_setup.c` and `constellation_setup.c`. We've placed these C source files in the `.../arch/arm/mach-ixp4xx` subdirectory of the kernel source tree. Listing 4-12 contains the complete makefile for this directory from a recent Linux release.

Listing 4-12

Makefile from `.../arch/arm/mach-ixp4xx` Kernel Subdirectory

```
#
# Makefile for the linux kernel.
#

obj-y += common.o common-pci.o

obj-$(CONFIG_ARCH_IXDP4XX) += ixdp425-pci.o ixdp425-setup.o
obj-$(CONFIG_MACH_IXDPG425) += ixdp425-pci.o coyote-setup.o
obj-$(CONFIG_ARCH_ADI_COYOTE) += coyote-pci.o coyote-setup.o
obj-$(CONFIG_MACH_GTWX5715) += gtwx5715-pci.o gtwx5715-setup.o
```

You might be surprised by the simplicity of this makefile. Much work has gone into the development of the kernel build system for just this reason. For the average developer who simply needs to add support for his custom hardware, the design of the kernel build system makes these kinds of customizations very straightforward.¹⁰

¹⁰ In actuality, the kernel build system is very complicated, but most of the complexity is cleverly hidden from the average developer. As a result, it is relatively easy to add, modify, or delete configurations without having to be an expert.

Looking at this makefile, it might be obvious what must be done to introduce new hardware setup routines conditionally based on your configuration options. Simply add the following two lines at the bottom of the makefile, and you're done:

```
obj-$(CONFIG_ARCH_VEGA) += vega_setup.o
obj-$(CONFIG_ARCH_CONSTELLATION) += constellation_setup.o
```

These steps complete the simple addition of setup modules specific to the hypothetical example custom hardware. Using similar logic, you should now be able to make your own modifications to the kernel configuration/build system.

4.3.7 Kernel Documentation

A wealth of information is available in the Linux source tree itself. It would be difficult indeed to read it all because there are nearly 650 documentation files in 42 subdirectories in the `.../Documentation` directory. Be cautious in reading this material: Given the rapid pace of kernel development and release, this documentation tends to become outdated quickly. Nonetheless, it often provides a great starting point from which you can form a foundation on a particular kernel subsystem or concept.

Do not neglect the Linux Documentation Project, found at www.tldp.org, where you might find the most up-to-date version of a particular document or man page.¹¹ The list of suggested reading at the end of this chapter duplicates the URL for the Linux Documentation Project, for easy reference. Of particular interest to the previous discussion is the Kbuild documentation found in the kernel `.../Documentation/kbuild` subdirectory.

No discussion of Kernel documentation would be complete without mentioning Google. One day soon, *Googling* will appear in Merriam Webster's as a verb! Chances are, many problems and questions you might ask have already been asked and answered before. Spend some time to become proficient in searching the Internet for answers to questions. You will discover numerous mailing lists and other information repositories full of useful information related to your specific project or problem. Appendix E contains a useful list of open-source resources.

¹¹ Always assume that features advance faster than the corresponding documentation, so treat the docs as a guide rather than indisputable facts.

4.4 Obtaining a Linux Kernel

In general, you can obtain an embedded Linux kernel for your hardware platform in three ways: You can purchase a suitable commercial embedded Linux distribution; you can download a free embedded distribution, if you can find one suitable for your particular architecture and processor; or you can find the closest open-source Linux kernel to your application and port it yourself. We discuss Linux porting in Chapter 16, “Porting Linux.”

Although porting an open source kernel to your custom board is not necessarily difficult, it represents a significant investment in engineering/development resources. This approach gives you access to free software, but deploying Linux in your development project is far from free, as we discussed in Chapter 1, “Introduction.” Even for a small system with minimal application requirements, you need many more components than just a Linux kernel.

4.4.1 What Else Do I Need?

This chapter has focused on the layout and construction of the Linux kernel itself. As you might have already discovered, Linux is only a small component of an embedded system based on Linux. In addition to the Linux kernel, you need the following components to develop, test, and launch your embedded Linux widget:

- Bootloader ported to and configured for your specific hardware platform
- Cross-compiler and associated toolchain for your chosen architecture
- File system containing many packages—binary executables and libraries compiled for your native hardware architecture/processor
- Device drivers for any custom devices on your board
- Development environment, including host tools and utilities
- Linux kernel source tree enabled for your particular processor and board

These are the components of an embedded Linux distribution.

4.5 Chapter Summary

- The Linux kernel is more than 10 years old and has become a main-stream, well-supported operating system for many architectures.
- The Linux open source home is found at www.kernel.org. Virtually every release version of the kernel is available there, going all the way back to Linux 1.0.
- We leave it to other great books to describe the theory and operation of the Linux kernel. Here we discussed how it is built and identified the components that make up the image. Breaking up the kernel into understandable pieces is the key to learning how to navigate this large software project.
- This chapter covered the kernel build system and the process of modifying the build system to facilitate modifications.
- Several kernel configuration editors exist. We chose one and examined how it is driven and how to modify the menus and menu items within. These concepts apply to all the graphical front ends.
- The kernel itself comes with an entire directory structure full of useful kernel documentation. This is a helpful resource for understanding and navigating the kernel and its operation.
- This chapter concluded with a brief introduction to the options available for obtaining an embedded Linux distribution.

4.5.1 Suggestions for Additional Reading

Linux Kernel HOWTO:

www.tldp.org/HOWTO/Kernel-HOWTO

Kernel Kbuild documentation:

<http://sourceforge.net/projects/kbuild/>

The Linux Documentation Project:

www.tldp.org/

Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification,
Version 1.2

TIS Committee, May 1995

Linux kernel source tree:

`.../Documentation/kbuild/makefiles.txt`

Linux kernel source tree:

`.../Documentation/kbuild/kconfig-language.txt`

Linux Kernel Development, 2nd Edition

Robert Love

Novell Press, 2005