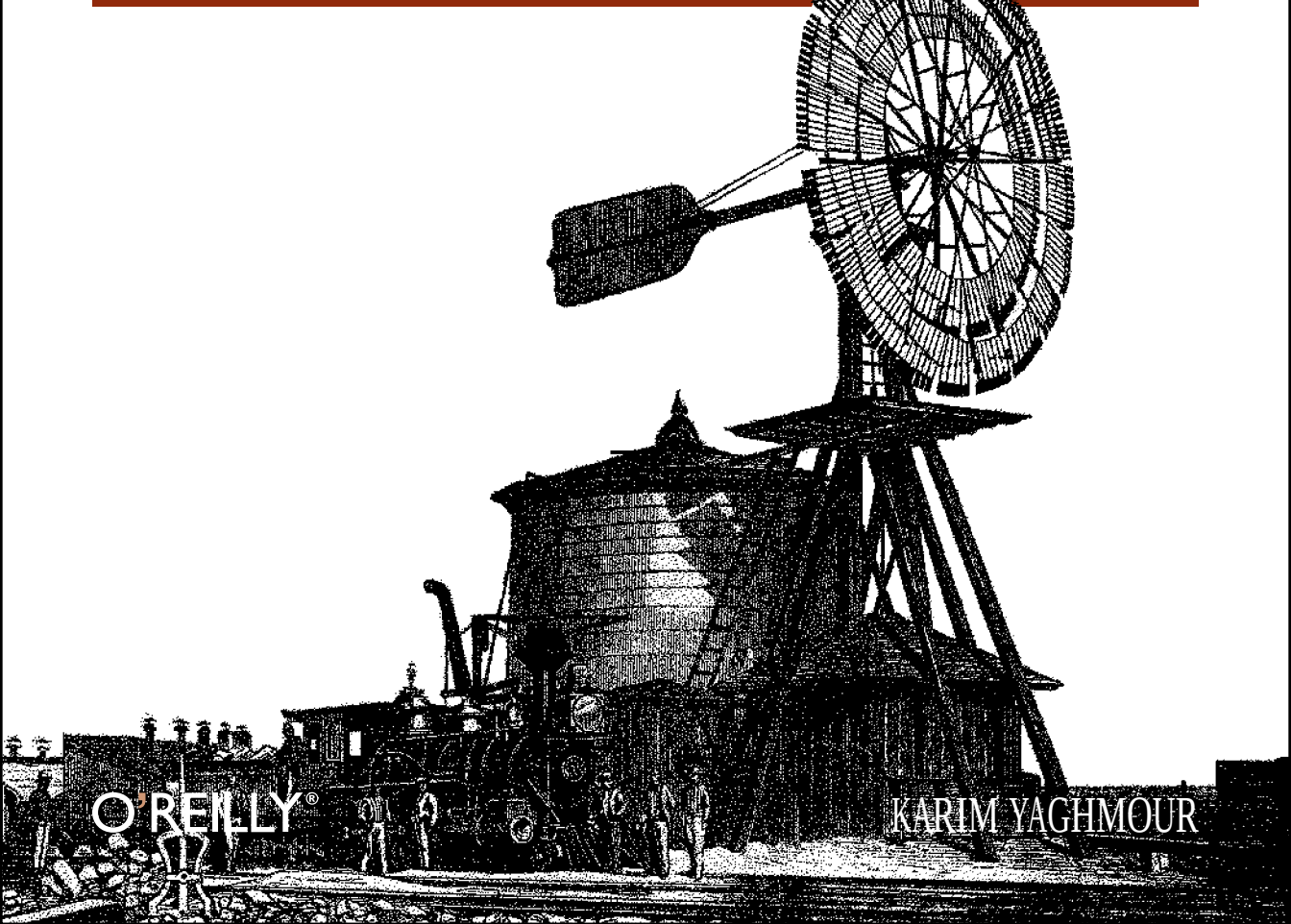
CONCEPTS, TECHNIQUES, TRICKS, AND TRAPS

Building Embedded LINUX SYSTEMS



O'REILLY®

KARIM YAGHMOUR

Building Embedded LINUX SYSTEMS

Related titles from O'Reilly

Building Secure Servers with Linux

Learning Red Hat Linux

Linux Device Drivers

Linux in a Nutshell

Linux Network Administrator's Guide

LPI Linux Certification in a Nutshell

Programming Embedded Systems in C and C++

Running Linux

Understanding the Linux Kernel

Also available

The Linux Web Server CD Bookshelf

Building Embedded LINUX SYSTEMS

Karim Yaghmour

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

CHAPTER 5

Kernel Considerations

The kernel is the central software component of all Linux systems. Its capabilities very much dictate the capabilities of the entire system. If the kernel you use fails to support one of your target's hardware components, for instance, this component will be useless as long as this specific kernel runs on your target.

Many books and online documentation already discuss the kernel's internals, its programming, its setup, and its use in user systems at length. I will not, therefore, cover these issues here. If you are interested in such issues, have a look at *Running Linux*, *Linux Device Drivers*, and *Understanding the Linux Kernel* by O'Reilly. These books cover the kernel's setup and use, its programming, and its internals, respectively. You may also want to take a look at the Linux Kernel HOWTO available from the LDP.

Our discussion will be limited to issues about the preparation of a Linux kernel for use in an embedded system. Specifically, we will discuss kernel selection, configuration, compilation, and installation. Each step will get us closer to the goal of obtaining a functional kernel with its related modules for our target system. Our discussion will end with coverage of the aspects of the kernel's operation that are specific to embedded systems.

Selecting a Kernel

Though there is only one main repository for the kernel, *http://www.kernel.org/*, the versions available from that site aren't always appropriate for all the architectures supported by Linux. In fact, these versions will often not even build for, much less run on, some of the most popular architectures in embedded Linux systems. This is primarily because the development of Linux for these architectures isn't synchronized with the main kernel releases.

To have a working kernel for your target, you need to obtain one of the versions made available by the development team in charge of your target's underlying processor architecture. Since each architecture is maintained by a different team, the site

of choice for a kernel varies accordingly. Table 5-1 provides a list of locations where you will find the most appropriate kernel for your architecture, along with the means of download available from that site.

Table 5-1. Most appropriate kernel location for each processor architecture

Processor architecture	Most appropriate kernel location	Available download means
x86	*http://www.kernel.org/*	ftp, http, rsync
ARM	*http://www.arm.linux.org.uk/developer/*	ftp, rsync
PowerPC	*http://penguinppc.org/*	ftp, http, rsync, bitkeeper
MIPS	*http://www.linux-mips.org/*	cvs
SuperH	*http://linuxsh.sourceforge.net/*	cvs
M68k	*http://www.linux-m68k.org/*	ftp, http

As you can see, most of these sites are the same ones I recommended for each architecture in Chapter 3. That said, these are not the only kernel locations for each target. Other locations may also provide versions for your target. To begin with, some of these sites have mirrors that provide the same content. Then there are the kernels provided by various individuals, companies, and organizations. Exercise caution if you intend to use the latter type of kernel, as these kernels may not be supported by the community* and you may be forced to rely on the provider's support, if available, in case of problems.

Once you have found the download site that is most appropriate for you, you will need to select a kernel version from that site. This is a difficult decision, as some versions have broken features, even if the same features were fully functional in older versions. The best way to find this sort of information is to stay in touch with the community maintaining the kernel for your architecture. This doesn't mean sending any emails or contacting anyone, but it involves subscribing to the appropriate mailing lists and keeping watch of the important notices on that list and on the port's main web site.

Some of these sites, such as the ARM site, don't necessarily distribute full kernels. Rather, they distribute patches to the official kernel. To obtain the appropriate kernel for your architecture, you must then download the kernel from the main repository and apply to it the appropriate patch provided by your port's site.

For our ARM-based user interfaces, we download plain 2.4.18 from *http://www.kernel.org/* and the 2.4.18-rmk5 patch from the official ARM Linux site, *http://www.arm.*

* This lack of support from the community won't necessarily be due to lack of code availability (which shouldn't happen since Linux is distributed under the terms of the GPL), but most likely because the modifications to the kernel's functionality made by that provider are understood only by her. It may also be that the kernel modifications are not considered mature enough, or even desirable, by the community to warrant inclusion in the main kernel tree.

Kernel Version Variations

The versions distributed by the alternative repositories often use variations on the kernel's versioning scheme to identify their work. Russell King, the maintainer of the ARM tree, distributes his kernels with the -rmk extension. Other developers base their work on Russell's work and add their own extensions. Nicolas Pitre, another ARM Linux developer, adds the -np extension to his kernels, and the maintainers of the handhelds.org Familiar distribution add the -hh extension to their kernels. Hence, kernel 2.4.20-rmk3-hh24, which I mentioned in Chapter 1, is handhelds.org's Release 24 of Russell's Release 3, which is itself based on Marcelo Tosatti's 2.4.20.

(Though Linus Torvalds is the usual maintainer of Linux releases, Linus passed the maintenance of the 2.4.x series on to Marcelo so he could concentrate on the 2.5.x development series.)

linux.org.uk/. By applying the rmk5 patch to the vanilla 2.4.18, we obtain the 2.4.18-rmk5 kernel, which contains all the features required for ARM-based systems.

Most of the time, the latest known-to-be-functional version is the best one to use. So if 2.4.17 and 2.4.18 are known to work on your target, 2.4.18 should be the preferable one. There are cases, however, in which this doesn't hold true. Most folks who follow the kernel's development are aware, for example, that Versions 2.4.10 to 2.4.15, inclusive, are to be avoided, because they were part of a period during which a lot of changes were being integrated into the kernel and are therefore sometimes unstable. Again, this is the sort of information you can obtain by keeping in touch with the appropriate mailing lists and web sites.

If you find it too time consuming to subscribe to your port's mailing list or to the main kernel mailing list, you owe it to yourself to at least visit your port's web site once a week and read the *Kernel Traffic* (*http://kt.zork.net/kernel-traffic/*) weekly newsletter. *Kernel Traffic* provides a summary of the most important discussions that occurred on the main kernel mailing list during the past week.

Once you have found the appropriate kernel version for your target, download it into the *${PRJROOT}/kernel* directory, extract it, and rename it if necessary, as we have done in the previous chapter in "Kernel Headers Setup." Renaming the kernel directory will avoid the mistake of overwriting it while extracting another kernel you might download in the future.

Whichever version you choose, do not refrain from trying a couple of different kernel versions for your target. In addition to the recommendations and bug reports seen on the Net, your evaluation of different versions will provide you with insight on your hardware's interaction with the kernel.

You may also want to try some of the various patches made available by some developers. Extra kernel functionality is often available as an independent patch before it

is integrated into the mainstream kernel. Robert Love's kernel preemption patch, for instance, was maintained as a separate patch before it was integrated by Linus into the 2.5 development series. We will discuss a few kernel patches in Chapter 11. Have a look at *Running Linux* (O'Reilly) if you are not familiar with patches.

Configuring the Kernel

Configuration is the initial step in the build of a kernel for your target. There are many ways to configure the kernel, and there are many options from which to choose .

Regardless of the configuration method you use or the actual configuration options you choose, the kernel will generate a *.config* file at the end of the configuration and will generate a number of symbolic links and file headers that will be used by the rest of the build.

We will limit our discussion to the aspects of kernel configuration that differ in embedded systems. You can consult the various references I mentioned earlier if you are not familiar with kernel configuration.

Configuration Options

It is during configuration that you will be able to select the options you want to see included in the kernel. Depending on your target, the option menus available will change, as will their content. Some options, however, will be available no matter which embedded architecture you choose. The following is the list of main menu options available to all embedded Linux architectures:

- Code maturity level options
- Loadable module support
- General setup
- Memory technology devices
- Block devices
- Networking options
- ATA/IDE/MFM/RLL support
- SCSI support
- Network device support
- Input core support
- Character devices
- Filesystems
- Console drivers
- Sound
- Kernel hacking

I will not give the details of each option, since the kernel configuration menu provides help capabilities you can refer to as you perform the configuration. Notice, however, that we discussed many of these options in Chapter 3.

One of the most important option menus is the one in which you choose the exact instance of the processor architecture that best fits your target. The name of this menu, however, varies according to your architecture. Table 5-2 provides the system and processor selection option menu name, along with the correct kernel architecture name for each. When issuing *make* commands, we need to set the `ARCH` variable to the architecture name recognized by the kernel Makefiles.

Table 5-2. System and processor selection option and kernel architecture name according to processor architecture

Processor architecture	System and processor selection option	Kernel architecture name
x86	Processor type and features	i386
ARM	System type	arm
PPC	Platform support	ppc
MIPS	Machine selection/CPU selection	mips or mips64[a]
SH	Processor type and features	sh
M68k	Platform-dependent support	m68k

[a] Depending on the CPU.

Some options are available only for certain architectures. Table 5-3 lists these options and indicates their availability for each architecture, as displayed by the kernel's configuration menus.

Table 5-3. Hardware support options for each architecture

Option	x86	ARM	PPC	MIPS	SH	M68k
Parallel port support	X	X		X		
IEEE 1394 support	X	X	X		X	
IrDA support	X	X	X	X		
USB support	X	X	X	X		
Bluetooth support	X	X	X			

Some architectures have their own specific configuration option menus. The following is a list of such menus for the ARM architecture:

- Acorn-specific block devices
- Synchronous serial interfaces
- Multimedia capabilities port drivers

Here is the list of menus specific to the PPC:

- MPC8xx CPM options
- MPC8260 communication options

The fact that an option is available in your architecture's configuration menu does not automatically mean that this feature is supported for your target. Indeed, the configuration menus may allow you to enable many kernel features that have never been tested for your target. There is no VGA console, for instance, on ARM systems. The configuration menu of the kernel, however, will allow you to enable support for the VGA console. In this case, the actual kernel build will fail if you enable support for this option. In other cases, the selected feature, or even the entire kernel, will not be functional. To avoid these types of problems, make sure the options you choose are supported for your target. Most of the time, as in the case of the VGA console, it is a matter of common sense. When the choice doesn't seem as evident, visiting the appropriate project web site, such as the ones provided in Chapter 3, will help you determine whether the feature is supported for your target.

In some cases, the fact that an option is not displayed in your architecture's configuration menu doesn't mean that this feature can't actually be used on your target. Many of the features listed in Table 5-3, such as Bluetooth, are mostly architecture independent, and should run on any architecture without a problem. They aren't listed in the configuration menus of certain architectures, because they've either not been tested on those architectures, or the maintainers of those ports or the maintainers of the feature haven't been asked to add the feature in the architecture's main *config.in* file.* Again, the resources listed in Chapter 3 are a good start for finding out about which unlisted features are possibly supported on your target.

Configuration Methods

The kernel supports four main configuration methods:

make config
: Provides a command-line interface where you are asked about each option one by one. If a *.config* configuration file is already present, it uses that file to set the default values of the options it asks you to set.

make oldconfig
: Feeds *config* with a an existing *.config* configuration file, and prompts you to configure only those options you had not previously configured. This contrasts with *make config*, which asks you about all options, even those you may have previously configured.

* *config.in* files control the options displayed in the configuration menus.

make menuconfig
: Displays a curses-based terminal configuration menu. If a *.config* file is present, it uses it to set default values, as with *make config*.

make xconfig
: Displays a Tk-based X Window configuration menu. If a *.config* file is present, it uses it to set default values, as with *make config* and *make menuconfig*.

Any of these can be used to configure the kernel. They all generate a *.config* file in the root directory of the kernel sources. (This is the file that contains the full detail of the options you choose.)

Few developers actually use the *make config* command to configure the kernel. Instead, most use *make menuconfig*. You can also use *make xconfig*. Keep in mind, however, that *make xconfig* may have some broken menus in some architectures; as is the case for the PowerPC, for instance.

To view the kernel configuration menu, type the appropriate command at the command line with the proper parameters. For our ARM-based user interface modules, we use the following command line:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

We then proceed to choose the configuration options appropriate to our target. Many features and drivers are available as modules and we can choose here whether to have them built in the kernel or whether to build them as modules. Once we are done configuring the kernel, we use the Escape key or select the Exit item to quit the configuration menu. We are then prompted by the configuration utility to confirm that we want to save the configuration. By choosing Yes, we save the kernel's configuration and create a *.config* file. In addition to creating the *.config* file, a few header files and symbolic links are created. If we choose No, the configuration is not saved and any existing configuration is left unmodified.

Apart from the main configuration options, some architectures, such as the PPC and the ARM, can be configured using custom tailored configurations for the various boards implemented using the architecture. In those cases, the defaults provided with the kernel will be used to generate the *.config* file. For example, here is how I configure the kernel for the TQM860L PowerPC board I have:

```
$ make ARCH=ppc CROSS_COMPILE=powerpc-linux- TQM860L_config
$ make ARCH=ppc CROSS_COMPILE=powerpc-linux- oldconfig
```

Managing Multiple Configurations

It is often desirable to test different configurations using the same kernel sources. By changing the kernel's configuration, however, we destroy the previous configuration, because all the configuration files are overwritten by the kernel's configuration utilities. To save a configuration for future use, we need to save the *.config* files created by the kernel's configuration. These files can later be reused to restore a previous kernel configuration.

The easiest way to back up and retrieve configurations is to use the kernel's own configuration procedures. The menus displayed by both the `menuconfig` and `xconfig` Makefile targets allow you to save and restore configurations. In each case, you need to provide an appropriate filename.

You can also save the *.config* files by hand. In that case, you need to copy the configuration file created by the kernel configuration utilities to an alternative location for future use. To use a saved configuration, you will need to copy the previously saved *.config* file back into the kernel's root directory and then use the *make* command with the `oldconfig` Makefile target to configure the kernel using the newly copied .config. As with the `menuconfig` Makefile target, the `oldconfig` Makefile target creates a few headers files and symbolic links.

Whether you copy the files manually or use the menus provided by the various utilities, store the configurations in an intuitive location and use a meaningful naming scheme for saving your configurations. Using our project layout, I suggest that you store all your configurations in the *${PRJROOT}/kernel* directory so that the configuration files may live independently from the actual kernel sources while still remaining with the other kernel-related material. To identify each configuration file, prepend each filename with the kernel version it relates to and a small descriptive comment or a date or both. Leave the *.config* extension as-is, nevertheless, to identify the file as a kernel configuration file.

In the case of the 2.4.18 kernel we are using, for instance, I tried a configuration where I disabled serial port support. I called the corresponding configuration file *2.4.18-no-serial.config*. I also maintain the latest known "best" configuration as *2.4.18.config*. Feel free to adopt the naming convention that is most intuitive for you, but you may want to avoid generic names such as *2.4.18-test1.config*.

Using the EXTRAVERSION Variable

If you are using multiple variants of the same kernel version, you will find the `EXTRAVERSION` variable to be quite useful in identifying each instance. The `EXTRAVERSION` variable is appended to the kernel's version number to provide the kernel being built with its final name. The rmk5 patch we applied on our plain 2.4.18, for example, sets `EXTRAVERSION` to -rmk5 and the resulting version for that kernel is 2.4.18-rmk5.

The final version number is also used to name the directory where the modules built for the kernel are stored. Hence, modules built for two kernels based on the same initial version but with different `EXTRAVERSION`s will be stored in two different directories, whereas modules built for two kernels based on the same initial version but that have no `EXTRAVERSION` will be stored in the same directory.

You can also use this variable to identify variants based on the same kernel version. To do so, edit the Makefile in the main kernel directory and set `EXTRAVERSION` to your desired value. You will find it useful to rename the directory containing this modified source code using this same value. If, for example, the `EXTRAVERSION` of a 2.4.18

kernel is set to `-motor-diff`, the parent directory should be named *2.4.18-motor-diff*. The naming of the backup *.config* files should also reflect the use of `EXTRAVERSION`. The configuration file for the kernel with disabled serial support should therefore be called *2.4.18-motor-diff-no-serial.config* in this case.

Compiling the Kernel

Compiling the kernel involves a number of steps: building the kernel dependencies, building the kernel image, and building the kernel modules. Each step uses a separate *make* command and is described separately in this section. However, you could also carry out all these steps using a single command line.

Building Dependencies

Most files in the kernel's sources depend on a number of header files. To build the kernel adequately, the kernel's Makefiles need to know about these dependencies. For each subdirectory in the kernel tree, a hidden *.depend* file is created during the dependencies build. This contains the list of header files that each file in the directory depends on. As with other software that relies on *make*, only the files that depend on a header that changed since the last build will need to be recompiled when a kernel is rebuilt.

From the kernel source's root directory, the following command builds the kernel's dependencies:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- clean dep
```

As in the configuration of the kernel earlier, we set the `ARCH` and `CROSS_COMPILE` variables. As I explained in Chapter 4, `CROSS_COMPILE` is only required when source code is actually compiled, and could be omitted here. On the other hand, we will need to set at least the `ARCH` variable for every *make* command we issue because we are cross-compiling the kernel. Even when issuing *make clean* or *make distclean*, we will need to set this variable. Otherwise, the kernel's Makefiles assume that the operations are to be carried out for the kernel code related to the host's architecture.

The `ARCH` variable indicates the architecture for which this kernel is built. This variable is used by the kernel Makefiles to choose which architecture-dependent directory is going to be used. When compiling the kernel for your target, you must set this variable to your target's architecture.

The `CROSS_COMPILE` variable is used by the kernel's Makefiles to construct the names of the tools used in the kernel's build. The name of the C compiler, for instance, is the result of the concatenation of the value of `CROSS_COMPILE` and the letters "gcc". In the case of our ARM target, the C compiler's final name is *arm-linux-gcc*, which is the actual name of the compiler we built for this target using the instructions in Chapter 4. This also explains why the trailing hyphen on the previous command line

is important. Without this hyphen, the Makefile would try to use the *arm-linuxgcc* compiler, which doesn't exist.

The building of the dependencies is relatively short. On my PowerBook, this takes two minutes. There are usually no errors possible at this stage. If you do see errors, the kernel you have probably suffers from fundamental problems.

Building the Kernel

With the dependencies built, we can now compile the kernel image:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

The `zImage` target instructs the Makefile to build a kernel image that is compressed using the gzip algorithm.* There are, nevertheless, other ways to build a kernel image. The `vmlinux` target instructs the Makefile to build only the uncompressed image. Note that this image is generated even when a compressed image is requested.

On the x86, there is also the `bzImage` target. The "bzImage" name stands for "big zImage," and has nothing to do with the *bzip2* compression utility. In fact, both the `bzImage` and `zImage` Makefile targets rely on the gzip algorithm. The difference between the two Makefile targets is that the compressed kernel images generated using `zImage` cannot be larger than 512 KB, while those generated using `bzImage` are not bound by this limit. If you want more information regarding the differences between `zImage` and `bzImage`, have a look at the *Documentation/i386/boot.txt* file included in the kernel sources.

If you chose any options not supported by your architecture during the kernel configuration or if some kernel option is broken, your build will fail at this stage. If all goes well, this should take a few minutes longer than the dependency build. On my hardware configuration, it takes five minutes.

Verifying the Cross-Development Toolchain

Notice that the kernel build is the first real test for the cross-development tools we built in the previous chapter. If the tools you built earlier compile a functional kernel successfully, all the other software should build perfectly. Of course, you will need to download the kernel you built to your target to verify its functionality, but the fact that it builds properly is already a positive sign.

* Though `zImage` is a valid Makefile target for all the architectures we discussed in depth in Chapter 3, there are other Linux architectures for which it isn't valid.

Building the Modules

With the kernel image properly built, we can now build the kernel modules:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- modules
```

The duration of this stage depends largely on the number of kernel options you chose to build as modules instead of having linked as part of the main kernel image. This stage is seldom longer than the build of the kernel image. As with the kernel image, if your configuration is inadequate for your target or if a feature is broken, this stage of the build may fail.

With both the kernel image and the kernel modules now built, we are ready to install them for our target. Before we do so, note that if you needed to clean up the kernel's sources and return them to their initial state prior to any configuration, dependency building, or compilation, you could use the following command:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- distclean
```

Be sure to backup your kernel configuration file prior to using this command, since *make distclean* erases all the files generated during the previous stages, including the *.config* file, all object files, and the kernel images.

Installing the Kernel

Ultimately, the kernel we generated and its modules will have to be copied to your target to be used. I will cover the actual copying of the kernel and its modules in Chapters 6 and 9. Meanwhile, we will discuss how to manage multiple kernel images and their corresponding module installations. The configuration of the target's boot layout and its root filesystem depend on the techniques we discuss below.

Managing Multiple Kernel Images

In addition to using separate directories for different kernel versions, you will find it useful to have access to multiple kernel images to test on your target. Since these images may be built using the same sources, we need to copy them out of the kernel source and into a directory where they can be properly identified. In our setup, the repository for these images is the *${PRJROOT}/images* directory.

For each kernel configuration, we will need to copy four files: the uncompressed kernel image, the compressed kernel image, the kernel symbol map, and the configuration file. The last three are found within the kernel sources' root directory and are called *vmlinux*, *System.map*, and *.config*, respectively. The compressed kernel image file is found in the *arch/YOUR_ARCH/boot* directory, where *YOUR_ARCH* is the name of your target's architecture, and is called *zImage* or *bzImage*, depending on the Makefile target you used earlier. For our ARM-based target, the compressed kernel image is *arch/arm/boot/zImage*.

Some architectures, such as the PPC, have many boot directories. In those cases, the kernel image to use is not necessarily the one located at *arch/YOUR_ARCH/boot/zImage*. In the case of the TQM board mentioned above, for example, the compressed kernel image that should be used is *arch/ppc/images/vmlinux.gz*. Have a look at the *arch/YOUR_ARCH/Makefile* for a full description of all the Makefile boot image targets for your architecture. In the case of the PPC, the type of boot image generated depends on the processor model for which the kernel is compiled.

To identify the four files needed, we use a naming scheme similar to that of the kernel's version. In the case of the kernel generated using *2.4.18-rmk5* sources, for instance, we copy the files as follows:

```
$ cp arch/arm/boot/zImage ${PRJROOT}/images/zImage-2.4.18-rmk5
$ cp System.map ${PRJROOT}/images/System.map-2.4.18-rmk5
$ cp vmlinux ${PRJROOT}/images/vmlinux-2.4.18-rmk5
$ cp .config ${PRJROOT}/images/2.4.18-rmk5.config
```

You could also include the configuration name in the filenames. So in the case of the kernel without serial support, for instance, we could name the four kernel files *zImage-2.4.18-rmk5-no-serial*, *System.map-2.4.18-rmk5-no-serial*, *vmlinux-2.4.18-rmk5-no-serial*, and *2.4.18-rmk5-no-serial.config*.

Installing Kernel Modules

The kernel Makefile includes the `modules_install` target for installing the kernel modules. By default, the modules are installed in the */lib/modules* directory. Since we are in a cross-development environment, however, we need to instruct the Makefile to install the modules in another directory.

As the kernel modules need to be used with the corresponding kernel image, we will install the modules in a directory with a name similar to that of the kernel image. So in the case of the 2.4.18-rmk5 kernel we are using, we install the modules in the *${PRJROOT}/images/modules-2.4.18-rmk5* directory. The content of this directory will later be copied to the target's root filesystem for use with the corresponding kernel on the target. To install the modules in that directory, we use:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- \
> INSTALL_MOD_PATH=${PRJROOT}/images/modules-2.4.18-rmk5 \
> modules_install
```

The `INSTALL_MOD_PATH` variable is prepended to the */lib/modules* path, so the modules are therefore installed in the *${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules* directory.

Once it is done copying the modules, the kernel tries to build the module dependencies needed for the module utilities during runtime. Since *depmod*, the utility that builds the module dependencies, is not designed to deal with cross-compiled modules, it will fail.

To build the module dependencies for your modules, you will need to use another module dependency builder provided with the BusyBox package. We will discuss BusyBox at length in Chapter 6. For now, download a copy of the BusyBox archive from *http://www.busybox.net/* into your *${PRJROOT}/sysapps* directory and extract it there.* From the BusyBox directory, copy the *scripts/depmod.pl* Perl script into the *${PREFIX}/bin* directory.

We can now build the module dependencies for the target:

```
$ depmod.pl \
> -k ./vmlinux -F ./System.map \
> -b ${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules > \
> ${PRJROOT}/images/modules-2.4.18-rmk5/lib/modules/2.4.18-rmk5/modules.dep
```

The *-k* option is used to specify the uncompressed kernel image, the *-F* option is used to specify the system map, and the *-b* option is used to specify the base directory containing the modules for which we need to build dependencies. Because the tool's output goes to the standard output, we redirect it to the actual dependency file, which is always called *modules.dep*.

In the Field

Let's take a look at the kernel's operation once it's installed on your target and ready to run. Because the algorithms and underlying source code is the same for embedded and regular systems, the kernel will behave almost exactly the same as it would on a workstation or a server. For this reason, the other books and online material on the subject, such as *Linux Device Drivers* and *Understanding the Linux Kernel* from O'Reilly, are much more appropriate for finding in-depth explanations of the kernel. There are, nevertheless, aspects particular to embedded Linux systems or that warrant particular emphasis.

Dealing with Kernel Failure

The Linux kernel is a very stable and mature piece of software. This, however, does not mean that it or the hardware it relies on never fail. *Linux Device Drivers* covers issues such as oops messages and system hangs. In addition to keeping these issues in mind during your design, you should think about the most common form of kernel failure: kernel panic.

When a fatal error occurs and is caught by the kernel, it will stop all processing and emit a kernel panic message. There are many reasons a kernel panic can occur. One of the most frequent is when you forget to specify to the kernel the location of its root filesystem. In that case, the kernel will boot normally and will panic upon trying to mount its root filesystem.

* Download BusyBox Version 0.60.5 or later.

The only means of recovery in case of a kernel panic is a complete system reboot. For this reason, the kernel accepts a boot parameter that indicates the number of seconds it should wait after a kernel panic to reboot. If you would like the kernel to reboot one second after a kernel panic, for instance, you would pass the following sequence as part of the kernel's boot parameters: `panic=1`.

Depending on your setup, however, a simple reboot may not be sufficient. In the case of our control module, for instance, a simple reboot may even be dangerous, since the chemical or mechanical process being controlled may get out of hand. For this reason, we need to change the kernel's panic function to notify a human operator who could then use emergency manual procedures to control the process. Of course, the actual panic behavior of your system depends on the type of application your system is used for.

The code for the kernel's panic function, `panic()`, is in the *kernel/panic.c* file in the kernel's sources. The first observation to be made is that the panic function's default output goes to the console.* Since your system may not even have a terminal, you may want to modify this function according to your particular hardware. An alternative to the terminal, for example, would be to write the actual error string in a special section of flash memory that is specifically set aside for this purpose. At the next reboot, you would be able to retrieve the text information from that flash section and attempt to solve the problem.

Whether you are interested in the actual text message or not, you can register your own panic function with the kernel. This function will be called by the kernel's panic function in the event of a kernel panic and can be used to carry out such things as signaling an emergency.

The list that holds the functions called by the kernel's own panic function is `panic_notifier_list`. The `notifier_chain_register` function is used to add an item to this list. Conversely, `notifier_chain_unregister` is used to remove an item from this list.

The location of your own panic function has little importance, but the registration of this function must be done during system initialization. In our case, we add a *mypanic.c* file in the *kernel/* directory of the kernel sources and modify that directory's Makefile accordingly. Here is the *mypanic.c* for our control module:

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/notifier.h>

static int my_panic_event(struct notifier_block *,
                          unsigned long,
                          void *);

static struct notifier_block my_panic_block = {
        notifier_call:   my_panic_event,
```

* The console is the main terminal to which all system messages are sent.

```
        next:            NULL,
        priority:        INT_MAX
};

int __init register_my_panic(void)
{
        printk("Registering buzzer notifier \n");

        notifier_chain_register(&panic_notifier_list,
                                &my_panic_block);

        return 0;
}

void ring_big_buzzer(void)
{
          …
}

static int my_panic_event(struct notifier_block *this,
                          unsigned long event,
                          void *ptr)
{
        ring_big_buzzer( );

        return NOTIFY_DONE;
}

module_init(register_my_panic);
```

The `module_init(register_my_panic);` statement ensures that the `register_my_panic` function is called during the kernel's initialization without requiring any modification of the kernel's startup functions. The registration function adds `my_panic_block` to the list of other blocks in the panic notifier list. The `notifier_block` structure has three fields. The first field is the function to be called, the second is a pointer to the next notifier block, and the third is the priority of this block. In our case, we want to have the highest possible priority. Hence the use of `INT_MAX`.

In case of kernel panic, `my_panic_event` is called as part of the kernel's notification of all panic functions. In turn, this function calls on `ring_big_buzzer`, which contains code to start a loud alarm to attract the human operator's attention to the imminent problem.