1

Software Considerations

YOU KNOW YOU WANT TO BUILD AN EMBEDDED APPLICATION, and you know you want to use Linux as the operating system. Where do you start?

With the hardware.

The hardware choices you make—processor, memory, flash, and so on—drive what you will do with the software. Because no software license cost is associated with Linux, most of your cost will be in the hardware itself. The more units you sell, the more true that is. Therefore, in any high-volume application, it's important to get the hardware right before you ever worry about the software. Linux has been adapted to many different microprocessors and microcontrollers, and more are supported all the time. Chances are that Linux already supports the processor you'll choose in some way. If not, and you have the time and expertise, you can support it yourself.

After you select the hardware you want to use, it's time to see how well Linux supports that hardware. So fire up your browser and search the Internet for Web sites devoted to `Linux + *your hardware device*`. Here are some good places:

- `www.google.com`
- `www.LinuxDevices.com`
- `www.LinuxHardware.net`
- `www.LinHardware.com`

After you've finally decided on your hardware platform, it's time to nail down exactly what you want the software to do. This chapter presents several software-related issues you'll need to consider.

Note: If you want to support a soft modem (also known as *winmodems*), you can write the driver for Linux and install it as a module at boot time. Linus doesn't like this, but admits that it doesn't violate the GPL.

Embedded Linux Toolkits

Porting Linux to a new hardware platform can be a daunting task. Fortunately, several embedded Linux toolkits are designed to simplify the job of building the binary that runs your device. Some toolkits, such as Lineo's Embedix and Monte Vista's Hard Hat Linux, are broadly focused, and are able to work on many processors for lots of different applications. Others are from smaller companies and focus on narrower processor sets and more limited application ranges. Still others, such as PeeWeeLinux, are not products distributed by companies, but rather projects built by a set of like-minded hackers in the traditional Open Source model.

Here are a few things to consider when you're looking for an embedded Linux toolkit:

- **Hardware support.** Does the Linux toolkit include support for the processor you want to use? If the toolkit doesn't already support that processor, does the toolkit vendor have the ability to develop that support quickly enough for your purposes? If so, how much will it cost? Does it fit in your budget?
- **Documentation.** Is the toolkit well documented? Are all of the programs involved documented? Does the documentation cover both high-level concepts such as architecture white papers and low-level documentation such as how to build binaries, how to add your software to the code base, and reference manuals to the build and runtime software?
- **Adaptability.** How adaptable is the embedded Linux toolkit to the particular application you're going to use? If it's a very narrow toolkit and you'll use it for several projects, how much work will be involved in changing the toolkit to the projects for which it's not as well suited?
- **Developer support.** What kind of developer support does the toolkit company offer, and how expensive is it? If you need a bug fixed, a question answered, or a device driver written, how quickly can you get a response from the company? Is a listserv available for the users of the product? If one is available, how active is it? Is anyone from the company answering questions on the list?
- **Field upgradeability.** Are facilities available in the toolkit for upgrading the software in the field? Does the toolkit company offer any means for delivering those upgrades, or is that left to you?

- **User interface.** If your application requires some sort of user interface, what are your options? Does the embedded Linux toolkit offer some sort of embedded video interface? If so, how much room does it take? If not, is there some sort of Web-based interface available for configuration?

  Note that the toolkit itself may not have any support for a graphical interface. Several Open Source projects and several commercial products are aimed at building small-footprint graphical user interfaces. For more information, see `http://www.linuxdevices.com/articles/AT9202043619.html`.
- **Track record.** Does the toolkit vendor have any examples of customers who have created a similar product out in the field? How successful was that vendor with the toolkit?

Kernel Features

The Linux kernel runs on a vast array of hardware architectures—everything from handhelds to mainframes. To support this sort of scalability, the kernel is highly configurable.

There are several ways of configuring the kernel (note that I'm using the word *configure* quite loosely here):

- Typing `make config`, `make menuconfig`, or `make xconfig` in the root of the kernel source runs the standard kernel configure routines. You can turn options on or off or sometimes compile them as modules so they can be loaded at runtime.
- There are hundreds—perhaps even thousands—of kernel patches floating around the Internet. Some are very small—enough to fix a small bug in one file. Medium-size patches may affect a half-dozen files and add support for a particular hardware device. Some large patches add or affect many dozens of files and add support for new architectures. Often applying a patch adds new questions or entire screens to the kernel-configuration screens previously described.
- The One True Method of really "configuring" the kernel to do exactly what you want is to hack on it yourself. Until recently, this was an exercise only for those who have lots of time and patience—the Linux kernel source code is well structured but somewhat obtuse. Linus doesn't believe in cluttering up the source with comments for the uninitiated (see Chapter 5 of Documentation/Coding Style in the Linux source tree).

Fortunately, times have changed and there are now several good overviews of the Linux kernel. Perhaps the most lucid is *Understanding the Linux Kernel* by Daniel Pierre Bovet and Marco Cesati (O'Reilly, 2000).

Networking, Filesystems, and Executable Formats

Some embedded Linux applications have no use for the networking code. Be sure to configure the kernel so that the networking code is skipped if you don't need it; it takes up a lot of space. Also, make sure that your kernel supports only the one or two filesystem types you actually need. Finally, you probably need only one executable format, ELF (Executable and Linking Format), for your embedded application, so be sure to turn off all the rest. For details on the ELF file format, consult the following documents:

- `http://ibiblio.org/pub/Linux/GCC/ELF.doc.tar.gz`
- `http://ibiblio.org/pub/Linux/GCC/elf.ps.gz`

In general, it's a good idea to look through the documentation for all the choices in the kernel build menu. If you're using `make menuconfig` to configure the kernel, you can press the question mark (`?`) at any time to get information on the choice you have highlighted.

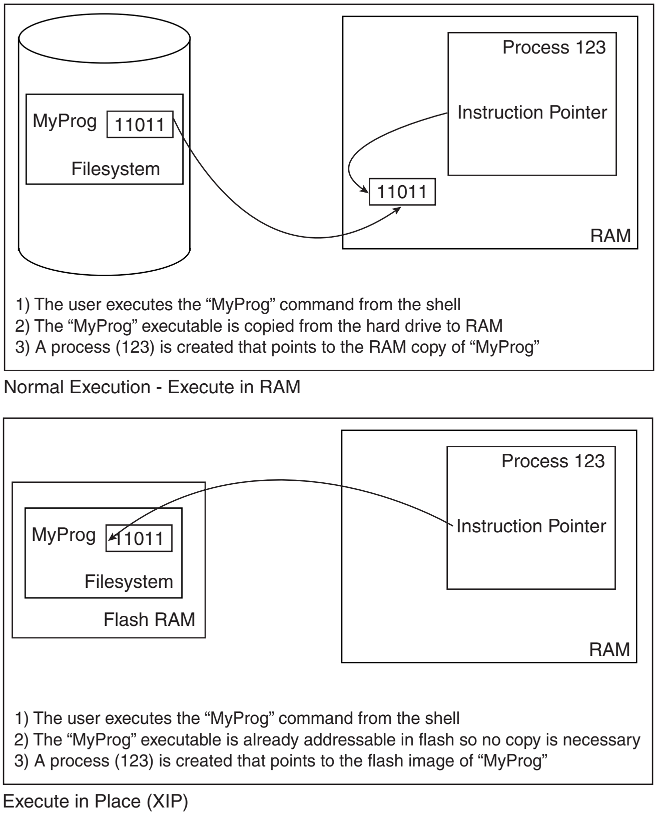
While you're developing your embedded device, it's handy to enable loadable module support in the kernel. That way, if you need support for a feature that you hadn't anticipated, you can go back, recompile the modules you need, copy them onto your device, and load them. Without loadable module support, you have to recompile the whole kernel, which can be a bit of a pain. When you're done developing the device, you can save some amount of RAM and ROM space by recompiling the kernel without loadable module support and with your drivers compiled directly into the kernel. However, you should figure out exactly how much of a savings this is and whether it's worth it—having loadable module support may be useful for upgrading drivers in the field.

Execute-in-Place (XIP)

One way you may be able to save a lot of RAM requirements is to run your executable programs directly from the long-term storage (ROM or flash) where they reside. This is called *execute-in-place* (or *XIP* for short).

On a desktop system, your application lives on the hard disk and must be read into RAM for execution. However, most embedded applications don't have a hard drive for long-term storage; instead, they have some sort of memory device, such as ROM or flash. Unlike a hard drive, these memory devices may be directly addressable by the CPU, like RAM. You can use the direct addressability of these memory devices to reduce your RAM requirements. Instead of copying the executable code from the memory device into RAM, an XIP system sets up the kernel structures that normally point into the RAM copy directly at the long-term storage locations. The code executes just as it would if it were copied to RAM. Depending on the speed of the processor, memory, and flash, there could be a performance penalty for using XIP.

Figure 1.1 shows the difference between normal execution and execution in place (XIP):



1) The user executes the "MyProg" command from the shell
2) The "MyProg" executable is copied from the hard drive to RAM
3) A process (123) is created that points to the RAM copy of "MyProg"

1) The user executes the "MyProg" command from the shell
2) The "MyProg" executable is already addressable in flash so no copy is necessary
3) A process (123) is created that points to the flash image of "MyProg"

Figure 1.1 XIP illustration.

Real-Time Operating Systems

Most of the time, it's not very important exactly how long a specific task takes to complete on a computer. For instance, when someone hits a key on the keyboard, they expect a letter to appear "instantly"—but how fast is "instantly"? It could be anywhere from a few tens of milliseconds to a couple of hundred milliseconds. You're probably not going to notice the difference between any two timings under 50 milliseconds (about 1/20th of a second). However, the timing of some operations in some computer applications is crucial. For instance, if your application moves a robotic arm to a precise location so that it can pick up a chip from a stack, then to another location so it can lay down the chip at a precise location on a PC board, the timing of the movement operations must be exact. This is when you need a real-time OS.

For instance, most modern cars have antilock braking systems. In these systems, special sensors detect when one or more wheels begin to "lock up"—a dangerous situation that can cause the vehicle (and its occupants!) to slide. In these situations, it's imperative that when the sensors detect a wheel beginning to lock, the braking on that wheel be reduced immediately. Have you ever worked on a computer system where you started a new program and the entire computer became unresponsive for several seconds? Imagine what would happen if the computer controlling your antilock brakes was similarly busy and unresponsive—right when a deer jumped in front of your car! This is a situation where you need what's called *hard real time*.

A hard real-time OS guarantees that, without fail, no matter what else happens, a response to an event will occur within a specified fixed time. For example, when wheel lockup is detected, braking for that wheel must be reduced within a certain number of milliseconds. In this example, a hard limit exists on how long the system can take to respond to the wheel lockup condition. This hard limit means that this is a hard real-time task—first, because there's an absolute limit on the amount of time available for a response, and second, because bad things will happen if the system fails to respond within the specified time limit. These two features of the task to be performed make it clear that the task requires a hard real-time operating system.

When working with a *soft* real-time OS, on the other hand, when an event occurs the system will try its best to service the event within an average response time. For example, when playing a game of Quake III, when a player fires a rocket at another player there's an expectation that the game program will draw a fiery explosion onscreen, make explosion noises, and dutifully subtract health from your opponent. With all of this complexity added to the existing events in the game, it's very likely that the frame rate of the game may drop slightly while rendering the explosion and playing back the additional audio, since these tasks require additional CPU time. If the frame rate should drop from 50 frames per second (fps) to, say, 40 fps for the duration of the explosion, no harm is done—the player continues to enjoy her game, since the system is still "fast enough." Being "fast enough" is a defining characteristic of soft real-time systems. In this case, no fixed frame rate is required of the system, and no harm occurs should the frame rate decrease slightly.

Both hard and soft real-time systems are useful, but they have distinctly different uses. In fact, a hard real-time OS usually accommodates tasks requiring both hard and soft real-time response.

Several attributes differentiate a real-time operating system from a standard operating system, as shown in the following list.

- **Response time predictability.** A hard real-time OS guarantees that the timing of some operations will be precise within its stated limitations. These response times are much faster than those of a typical operating system—they're measured in tens of microseconds (millionths of a second) instead of milliseconds (thousandths of a second).

- **Schedulability.** In a hard real-time operating system, a process can be scheduled to perform at a very precise time in the future or at a very precise interval. Again, the precision is down to the microsecond level instead of the millisecond level.
- **Stability under pressure.** In a hard real-time system, the processor can become inundated with far more signals from different sources than it can handle; however, some of those signals are much more important than other signals and must be recognized and dealt with. The ability to prioritize a vast array of different signals quickly and efficiently is another hallmark of a good real-time system.

For more details on embedded real-time systems, see Raj Rajkumar, et al, *The Concise Handbook of Linux for Embedded Real-Time Systems Version 1.0* (Pittsburgh, Pennsylvania: TimeSys Corporation, 2000), p. 5. In addition, `www.RealTimeLinux.org` provides a lot of information about real-time Linux.

Several real-time Linux kernel projects are underway, as shown in Table 1.1.

Table 1.1 **Real-Time Linux Projects**

Address	**Title**	**Site Sponsor**
`www.rtlinux.org`	RTLinux—Real time Linux	New Mexico Institute of Technology
`http://server.aero.polimi.it/projects/rtai/`	RTAI—Real Time Application Interface	Dipartimento di Ingegneria Aerospaziale Politecnico di Milano
`www.ittc.ukans.edu/kurt/`	KURT—The KU Real Time Linux	University of Kansas
`http://linux.ece.uci.edu/RED-Linux/index.html`	RED-Linux	University of California, Irvine

Creating or Acquiring a Development Environment

Before you can even start coding, you must either create or acquire a development environment for your chosen microprocessor or microcontroller. You'll need a C compiler, assembler (part of the compiler), linker, runtime library, debugging tools, and perhaps an emulator. The fastest way to acquire a development environment is to get it from one of the embedded Linux toolkits. Usually this is free or a nominal price.

Booting the Kernel

How the computer loads the operating system into memory and starts it is an issue that most software developers never have to think about. Most of us work on PCs or similar platforms that have a BIOS that does the dirty work of setting up the computer's hardware and finding and loading the OS loader (for example, LILO or GRUB) into RAM so that the kernel can start. The most we ever have to think about is which OS loader to use and how to configure it properly.

Welcome to the world of embedded devices, where you may start with a manual that says only something like this:

> "Hard Reset (HRESET)—Input" causes the hard reset exception to be taken and the physical address of the handler is always x'FFF00100'.
>
> *PowerPC 601 RISC Microprocessor User's Manual* (IBM Corporation, 1993), p. 5-16.

It's now up to *you* to write the assembly code to do the following:

1. Initialize all of the hardware.
2. Move the OS loader into memory from storage (or perhaps you just load the OS itself).
3. Jump into the code you just loaded.

Fortunately, most development boards for the various microcontrollers and microprocessors come with ample documentation and sample code for startup. There are also many examples on the Internet for the many different processors that Linux supports.

Software Size

Unlike a general-purpose computer system, an embedded system only requires enough software to actually get a specific set of jobs done. This means you can do without a lot of the fluff that normally goes into a general-purpose computer system such as X windows, email and newsreaders, games, and so on. By doing this, you can make the software image much smaller. This is important because you normally don't have a hard drive in an embedded application. Even when you do have one for storage, it may be better to put the system software and applications in ROM or flash so they aren't as vulnerable to corruption. Both ROM and flash are much more expensive byte-for-byte than space on a hard drive, so it's important to include just the software that your embedded application actually uses.

Another reason to limit the software that ships in your embedded application is simple; if it's not there, it can't break. It's usually very inconvenient to upgrade the software in an embedded application—sometimes it's impossible.

However, a tradeoff exists between engineering time and footprint size. Generally, the longer you engineer the product, the smaller and more efficient you can make it, reducing your memory requirements, and thus reducing the cost of each unit. However, the more time you spend on efficiency, the fewer features the product will have, given constant engineering resources. Also, if the product sits in engineering too long, you may lose considerable market advantage to your competitors.

Reducing the Software Footprint

So how do you reduce the memory/storage requirements of your embedded Linux application? Generally, there are three ways:

- Include only the software you need.
- Compile the software to reduce size.
- Compress the resulting software.

Chapter 2, "Minimal Linux," details how to reduce the size of your embedded Linux application.

Upgrading the Software in Place

Most software systems must be upgraded at some point in their lifetime for a variety of reasons: bugs and security vulnerabilities are found and fixed, new features are required, and so on. Embedded applications are no exception. However, if upgradeability is not designed in from the beginning, the upgrade process will be difficult or even impossible. For instance, unless you want your customers to do minor surgery on your embedded device, you shouldn't ship the software in ROM. The only way to replace software that's burned into ROM is to replace the ROM chip. Also, if you only have enough memory-plus-storage in the device to actually run the application, upgrading is very dangerous because there's no good place to put the software. You can download the new code over the old code, but one small error and your customer now owns a doorstop instead of an embedded Linux application.

So how do you design your upgrade process from the beginning?

1. Start by putting only the most low-level code in ROM and putting the rest in flash. The code that's in ROM will probably never be replaced, so you have to get it right. You may want to put the upgrade code itself in ROM; that way, no matter what happens in the field, the user can "upgrade" to a working set of code even after messing up an upgrade attempt.
2. Configure the machine with enough storage or RAM to hold two complete copies of the software in the machine simultaneously. Unless you put the upgrade software in ROM as described above, your application will become a doorstop if a catastrophic event occurs, such as loss of power. You want to reduce this "doorstop time" to the minimum possible. For instance, if your application

downloads the new software directly on top of the old software and this process takes 20 seconds, you have a doorstop time of 20 seconds. If the customer's power is lost, a cable is kicked loose, or the cat jumps onto the keyboard any time during this 20 seconds, your customer now has a doorstop—plus, they're angry, and it will cost you money.

Now imagine that you have 2^{1}/2 times the amount of flash that you really need in the device (the extra ^{1}/2 is for growth). Instead of overwriting the old code with the new, you store it in the extra room in the flash as it's downloading. When all the code is safely written to flash, you then change a few pointers to complete the update. Now your "doorstop time" is just a few milliseconds, while the pointers are updated in flash.

This procedure has other advantages. After all the new software has come across, you can make sure that the binary image is intact by including some checksum code. You can also make sure that the user has downloaded the right thing and has a later version than the one currently installed. None of this is possible if you overwrite the current software with the new as the download occurs.

3. Put all of the configuration information for your machine in a single place, such as a single file in flash. This makes all your software much easier to manage. It makes the upgrade process easier because there's only one file to manage (and perhaps change) during the upgrade process. Remember that you may have several versions out in the field, however, and you don't want to force users to move from a very old version to the newest version by upgrading through each version in between. If your software changes a lot between releases, the permutations can become enormous quickly, so keeping things as simple as possible helps the user avoid losing configuration information.

Of course, there are many ways to accomplish this objective. The process described above is more costly on a per-unit basis than simply putting everything in flash memory. Depending on your application and economies, it may make more sense to put everything in flash—just realize that in some instances you could end up with a lot of doorstops out in the field.