Mastering Embedded Linux Programming

Mastering Embedded Linux Programming takes you through the product cycle and gives you an in-depth description of the components and options that are available at each stage. You will begin by learning about toolchains, bootloaders, the Linux kernel, and how to configure a root filesystem. You will then learn how to use the two most commonly used build systems, Buildroot and the Yocto Project, to speed up and simplify the development process. Building on this solid base, the next section considers how to make best use of raw NAND/NOR flash memory and managed flash eMMC chips, including mechanisms for increasing the lifetime of the devices and to perform reliable in-field updates. We will then see the usage of POSIX threads, which have a big impact on the responsiveness and performance of the final device. The closing sections look at the techniques available for profiling and tracing applications and kernel code using perf and ftrace.

What you will learn from this book

- Understand the role of the Linux kernel and select an appropriate role for your application
- Use Buildroot and the Yocto Project to create embedded Linux systems quickly and efficiently
- Create customized bootloaders using U-Boot
- Employ perf and ftrace to identify performance bottlenecks
- Understand device trees and make changes to accommodate new hardware on your device
- Write applications that interact with Linux device drivers
- Design and write multi-threaded applications using POSIX threads
- Measure real-time latencies and tune the Linux kernel to minimize them

Who this book is written for

This book is ideal for Linux developers and system programmers who are already familiar with embedded systems and who want to know how to create best-in-class devices. A basic understanding of C programming and experience with systems programming is needed.

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Starting Out'
- A synopsis of the book’s content
- More information on Mastering Embedded Linux Programming
Chris Simmonds is a software consultant and trainer who lives in southern England. He has been using Linux in embedded systems since the late 1990s, during which he has worked on many interesting projects, including a stereoscopic camera, intelligent weighing scales, various set-top boxes and home routers, and even a large walking robot.

He is a frequent presenter at open source and embedded conferences, including the Embedded Linux Conference, Embedded World, and the Android Builders' Summit. He has been conducting training courses and workshops in embedded Linux since 2002 and in embedded Android since 2010. He has delivered hundreds of sessions to many well-known companies. You can see some of his work on the "Inner Penguin" blog at www.2net.co.uk.
Preface

An embedded system is a device with a computer inside that doesn't look like a computer. Washing machines, televisions, printers, cars, aircraft, and robots are all controlled by a computer of some sort, and in some cases, more than one. As these devices become more complex, and as our expectations of the things that we can do with them expand, the need for a powerful operating system to control them grows. Increasingly, Linux is the operating system of choice.

The power of Linux stems from its open source model, which encourages sharing of code. This means that software engineers from many backgrounds, and often employed by competing companies, can cooperate to create an operating system kernel that is up-to-date and tracks the development of the hardware. From this one code base, there is support from the largest super computers down to a wristwatch. Linux is only one component of the operating system. Many other components are needed to create a working system, from basic tools, such as a command shell, to graphical user interfaces, with web content and communicating with cloud services. The Linux kernel together with an extensive range of other open source components allow you to build a system that can function in a wide range of roles.

However, flexibility is a double-edged sword. While it gives a system designer a wide choice of solutions to a particular problem, it also presents the problem of knowing which are the best choices. The propose of this book is to describe in detail how to construct an embedded Linux system using free, open source projects to produce a robust, reliable, and efficient system. It is based on the experience of the author as a consultant and trainer over a period of many years, using examples to illustrate best practices.
What this book covers

Mastering Embedded Linux Programming is organized along the lines of the life cycle of a typical embedded Linux project. The first six chapters tell you what you need to know about how to set up the project and how a Linux system is put together, culminating in selecting an appropriate Linux build system. Next, comes the stage where certain key decisions must be made about the system architecture and design choices, including flash memory, device drivers, and the init system. Following this is the phase of writing applications to make use of the embedded platform you have built, and for which there are two chapters on processes, threads, and memory management. Finally, we come to the stage of debugging and optimizing the platform, which is discussed in chapters 12 and 13. The last chapter describes how to configure Linux for real-time applications.

Chapter 1, Starting Out, sets the scene by describing the choices available to the system designer at the start of a project.

Chapter 2, Learning About Toolchains, describes the components of a toolchain with an emphasis on cross-compiling. It describes where to get a toolchain and provides details on how to build one from the source code.

Chapter 3, All About Bootloaders, explains the role of the bootloader to initialize the hardware of the device and uses U-Boot and Bareboot as examples. It also describes the device tree, which is a means of encoding the hardware configuration, used in many embedded systems.

Chapter 4, Porting and Configuring the Kernel, provides information on how to select a Linux kernel for an embedded system and configure it for the hardware within the device. It also covers how to port Linux to the new hardware.

Chapter 5, Building a Root Filesystem, introduces the ideas behind the user space part of an embedded Linux implementation by means of a step-by-step guide on how to configure a root filesystem.

Chapter 6, Selecting a Build System, covers two embedded Linux build systems, which automate the steps described in the previous four chapters and conclude the first section of the book.

Chapter 7, Creating a Storage Strategy, discusses the challenges created by managing flash memory, including raw flash chips and embedded MMC or eMMC packages. It describes the filesystems that are applicable to each type of technology. It also covers techniques on how to update the device firmware in the field.
Chapter 8, *Introducing Device Drivers*, describes how kernel device drivers interact with the hardware with worked examples of a simple driver. It also describes the various ways of calling device drivers from the user space.

Chapter 9, *Starting up - the init Program*, shows how the first user space program, `init`, which starts the rest of the system. It describes the three versions of the `init` program, each suitable for a different group of embedded systems, with increasing complexity from BusyBox `init` to systemd.

Chapter 10, *Learning About Processes and Threads*, describes embedded systems from the point of view of the application programmer. This chapter looks at processes and threads, inter-process communication, and scheduling policies.

Chapter 11, *Managing Memory*, introduces the ideas behind virtual memory and how the address space is divided into memory mappings. It also covers how to detect memory that is being used and memory leaks.

Chapter 12, *Debugging with GDB*, shows you how to use the GNU debugger, GDB, to interactively debug both the user space and kernel code. It also describes the kernel debugger, `kdb`.

Chapter 13, *Profiling and Tracing*, covers the techniques available to measure the system performance, starting from whole system profiles and then zeroing in on particular areas where bottlenecks are causing poor performance. It also describes Valgrind as a tool to check the correctness of an application's use of thread synchronization and memory allocation.

Chapter 14, *Real-time Programming*, provides a detailed guide to real-time programming on Linux, including the configuration of the kernel and the real-time kernel patch, and also provides a description of tools to measure real-time latencies. It also covers information on how to reduce the number of page faults by locking the memory.
You are about to begin working on your next project, and this time it is going to be running Linux. What should you think about before you put finger to keyboard? Let's begin with a high-level look at embedded Linux and see why it is popular, what are the implications of open source licenses, and what kind of hardware you will need to run Linux.

Linux first became a viable choice for embedded devices around 1999. That was when Axis (www.axis.com) released their first Linux-powered network camera and TiVo (www.tivo.com) their first DVR (Digital Video Recorder). Since 1999, Linux has become ever more popular, to the point that today it is the operating system of choice for many classes of product. At the time of writing, in 2015, there are about two billion devices running Linux. That includes a large number of smartphones running Android, which uses a Linux kernel, and hundreds of millions of set top boxes, smart TVs, and Wi-Fi routers, not to mention a very diverse range of devices such as vehicle diagnostics, weighing scales, industrial devices, and medical monitoring units that ship in smaller volumes.

So, why does your TV run Linux? At first glance, the function of a TV is simple: it has to display a stream of video on a screen. Why is a complex Unix-like operating system like Linux necessary?

The simple answer is Moore's Law: Gordon Moore, co-founder of Intel, observed in 1965 that the density of components on a chip will double about every two years. That applies to the devices that we design and use in our everyday lives just as much as it does to desktops, laptops, and servers. At the heart of most embedded devices is a highly integrated chip that contains one or more processor cores and interfaces with main memory, mass storage, and peripherals of many types. This is referred to as a System on Chip, or SoC, and they are increasing in complexity in accordance with Moore's Law. A typical SoC has a technical reference manual that stretches to thousands of pages. Your TV is not simply displaying a video stream as the old analog sets used to do.
The stream is digital, possibly encrypted, and it needs processing to create an image. Your TV is (or soon will be) connected to the Internet. It can receive content from smartphones, tablets, and home media servers. It can be (or soon will be) used to play games. And so on and so on. You need a full operating system to manage this degree of complexity.

Here are some points that drive the adoption of Linux:

- Linux has the necessary functionality. It has a good scheduler, a good network stack, support for USB, Wi-Fi, Bluetooth, many kinds of storage media, good support for multimedia devices, and so on. It ticks all the boxes.
- Linux has been ported to a wide range of processor architectures, including some that are very commonly found in SoC designs — ARM, MIPS, x86, and PowerPC.
- Linux is open source, so you have the freedom to get the source code and modify it to meet your needs. You, or someone working on your behalf, can create a board support package for your particular SoC board or device. You can add protocols, features, and technologies that may be missing from the mainline source code. You can remove features that you don't need to reduce memory and storage requirements. Linux is flexible.
- Linux has an active community; in the case of the Linux kernel, very active. There is a new release of the kernel every 10 to 12 weeks, and each release contains code from around 1,000 developers. An active community means that Linux is up to date and supports current hardware, protocols, and standards.
- Open source licenses guarantee that you have access to the source code. There is no vendor tie-in.

For these reasons, Linux is an ideal choice for complex devices. But there are a few caveats I should mention here. Complexity makes it harder to understand. Coupled with the fast moving development process and the decentralized structures of open source, you have to put some effort into learning how to use it and to keep on re-learning as it changes. I hope that this book will help in the process.
Selecting the right operating system

Is Linux suitable for your project? Linux works well where the problem being solved justifies the complexity. It is especially good where connectivity, robustness, and complex user interfaces are required. However it cannot solve every problem, so here are some things to consider before you jump in:

• Is your hardware up to the job? Compared to a traditional RTOS (real-time operating system) such as VxWorks, Linux requires a lot more resources. It needs at least a 32-bit processor, and lots more memory. I will go into more detail in the section on typical hardware requirements.

• Do you have the right skill set? The early parts of a project, board bring-up, require detailed knowledge of Linux and how it relates to your hardware. Likewise, when debugging and tuning your application, you will need to be able to interpret the results. If you don’t have the skills in-house you may want to outsource some of the work. Of course, reading this book helps!

• Is your system real-time? Linux can handle many real-time activities so long as you pay attention to certain details, which I will cover in detail in Chapter 14, Real-time Programming.

Consider these points carefully. Probably the best indicator of success is to look around for similar products that run Linux and see how they have done it; follow best practice.

The players

Where does open source software come from? Who writes it? In particular, how does this relate to the key components of embedded development — the toolchain, bootloader, kernel, and basic utilities found in the root filesystem?

The main players are:

• The open source community. This, after all, is the engine that generates the software you are going to be using. The community is a loose alliance of developers, many of whom are funded in some way, perhaps by a not-for-profit organization, an academic institution, or a commercial company. They work together to further the aims of the various projects. There are many of them, some small, some large. Some that we will be making use of in the remainder of this book are Linux itself, U-Boot, BusyBox, Buildroot, the Yocto Project, and the many projects under the GNU umbrella.
Starting Out

- CPU architects—These are the organizations that design the CPUs we use. The important ones here are ARM/Linaro (ARM-based SoCs), Intel (x86 and x86_64), Imagination Technologies (MIPS), and Freescale/IBM (PowerPC). They implement or, at the very least, influence support for the basic CPU architecture.

- SoC vendors (Atmel, Broadcom, Freescale, Intel, Qualcomm, TI, and many others)—They take the kernel and toolchain from the CPU architects and modify it to support their chips. They also create reference boards: designs that are used by the next level down to create development boards and working products.

- Board vendors and OEMs—these people take the reference designs from SoC vendors and build them in to specific products, for instance set-top-boxes or cameras, or create more general purpose development boards, such as those from Avantech and Kontron. An important category are the cheap development boards such as BeagleBoard/BeagleBone and Raspberry Pi that have created their own ecosystems of software and hardware add-ons.

These form a chain, with your project usually at the end, which means that you do not have a free choice of components. You cannot simply take the latest kernel from kernel.org, except in a few rare cases, because it does not have support for the chip or board that you are using.

This is an ongoing problem with embedded development. Ideally, the developers at each link in the chain would push their changes upstream, but they don't. It is not uncommon to find a kernel which has many thousands of patches that are not merged upstream. In addition, SoC vendors tend to actively develop open source components only for their latest chips, meaning that support for any chip more than a couple of years old will be frozen and not receive any updates.

The consequence is that most embedded designs are based on old versions of software. They do not receive security fixes, performance enhancements, or features that are in newer versions. Problems such as Heartbleed (a bug in the OpenSSL libraries) and Shellshock (a bug in the bash shell) go unfixed. I will talk more about this later in this chapter under the topic of security.

What can you do about it? First, ask questions of your vendors: what is their update policy, how often do they revise kernel versions, what is the current kernel version, what was the one before that? What is their policy for merging changes up-stream? Some vendors are making great strides in this way. You should prefer their chips.

Secondly, you can take steps to make yourself more self-sufficient. This book aims to explain the dependencies in more detail and show you where you can help yourself. Don't just take the package offered to you by the SoC or board vendor and use it blindly without considering the alternatives.
Project lifecycle

This book is divided into four sections that reflect the phases of a project. The phases are not necessarily sequential. Usually they overlap and you will need to jump back to revisit things that were done previously. However, they are representative of a developer’s preoccupations as the project progresses:

- **Elements of embedded Linux** (chapters 1 to 6) will help you set up the development environment and create a working platform for the later phases. It is often referred to as the "board bring-up" phase.
- **System architecture and design choices** (chapters 7 to 9) will help you to look at some of the design decisions you will have to make concerning the storage of programs and data, how to divide work between kernel device drivers and applications, and how to initialize the system.
- **Writing embedded applications** (chapters 10 and 11) show how to make effective use of the Linux process and threads model and how to manage memory in a resource-constrained device.
- **Debugging and optimizing performance** (chapters 12 and 13) describe how to trace, profile, and debug your code in both the applications and the kernel.

The fifth section on real-time (Chapter 14, *Real-time Programming*) stands somewhat alone because it is a small, but important, category of embedded systems. Designing for real-time behavior has an impact on each of the four main phases.

The four elements of embedded Linux

Every project begins by obtaining, customizing, and deploying these four elements: the toolchain, the bootloader, the kernel, and the root filesystem. This is the topic of the first section of this book:

- **Toolchain**: This consists of the compiler and other tools needed to create code for your target device. Everything else depends on the toolchain.
- **Bootloader**: This is necessary to initialize the board and to load and boot the Linux kernel.
- **Kernel**: This is the heart of the system, managing system resources and interfacing with hardware.
- **Root filesystem**: This contains the libraries and programs that are run once the kernel has completed its initialization.
Of course, there is also a fifth element, not mentioned here. That is the collection of programs that are specific to your embedded application which make the device do whatever it is supposed to do, be it weigh groceries, display movies, control a robot, or fly a drone.

Typically you will be offered some or all of these elements as a package when you buy your SoC or board. But, for the reasons mentioned in the preceding paragraph, they may not be the best choices for you. I will give you the background to make the right selections in the first six chapters and I will introduce you to two tools that automate the whole process for you: Buildroot and the Yocto Project.

Open source

The components of embedded Linux are open source, so now is a good time to consider what that means, why open sources work the way they do and how this affects the often proprietary embedded device you will be creating from it.

Licenses

When talking about open source, the word, "free" is often used. People new to the subject often take it to mean nothing to pay, and open source software licenses do indeed guarantee that you can use the software to develop and deploy systems for no charge. However, the more important meaning here is freedom, since you are free to obtain the source code and modify it in any way you see fit and redeploy it in other systems. These licenses give you this right. Compare that with shareware licenses which allow you to copy the binaries for no cost but do not give you the source code, or other licenses that allow you to use the software for free under certain circumstances, for example, for personal use but not commercial. These are not open source.

I will provide the following comments in the interest of helping you understand the implications of working with open source licenses, but I would like to point out that I am an engineer and not a lawyer. What follows is my understanding of the licenses and the way they are interpreted.

Open source licenses fall broadly into two categories: the GPL (General Public License) from the Free Software Foundation and the permissive licenses derived from BSD (Berkeley Software Distribution), the Apache Foundation, and others.

The permissive licenses say, in essence, that you may modify the source code and use it in systems of your own choosing so long as you do not modify the terms of the license in any way. In other words, with that one restriction, you can do with it what you want, including building it into possibly proprietary systems.
Chapter 1

The GPL licenses are similar, but have clauses which compel you to pass the rights to obtain and modify the software on to your end users. In other words you share your source code. One option is to make it completely public by putting it onto a public server. Another is to offer it only to your end users by means of a written offer to provide the code when requested. The GPL goes further to say that you cannot incorporate GPL code into proprietary programs. Any attempt to do so would make the GPL apply to the whole. In other words, you cannot combine GPL and proprietary code in one program.

So, what about libraries? If they are licensed with the GPL, any program linked with them becomes GPL also. However, most libraries are licensed under the Lesser General Public License (LGPL). If this is the case, you are allowed to link with them from a proprietary program.

All of the preceding description relates specifically to GPL v2 and LGPL v2.1. I should mention the latest versions of GPL v3 and LGPL v3. These are controversial, and I will admit that I don't fully understand the implications. However, the intention is to ensure that the GPLv3 and LGPL v3 components in any system can be replaced by the end user, which is in the spirit of open source software for everyone. It does pose some problems though. Some Linux devices are used to gain access to information according to a subscription level or another restriction, and replacing critical parts of the software may compromise that. Set-top boxes fit into this category. There are also issues with security. If the owner of a device has access to the system code, then so might an unwelcome intruder. Often the defense is to have kernel images that are signed by an authority, the vendor, so that unauthorized updates are not possible. Is that an infringement of my right to modify my device? Opinions differ.

The TiVo set-top box is an important part of this debate. It uses a Linux kernel, which is licensed under GPL v2. TiVo release the source code of their version of the kernel and so comply with the license. TiVo also have a bootloader that will only load a kernel binary that is signed by them. Consequently, you can build a modified kernel for a TiVo box, but you cannot load it on the hardware. The FSF take the position that this is not in the spirit of open source software and refer to this procedure as "tivoization". The GPL v3 and LGPL v3 were written to explicitly prevent this happening. Some projects, the Linux kernel in particular, have been reluctant to adopt the version three licenses because of the restrictions it would place on device manufacturers.
Starting Out

Hardware for embedded Linux

If you are designing or selecting hardware for an embedded Linux project, what do you look out for?

Firstly, a CPU architecture that is supported by the kernel – unless you plan to add a new architecture yourself of course! Looking at the source code for Linux 4.1, there are 30 architectures, each represented by a sub-directory in the arch/ directory. They are all 32- or 64-bit architectures, most with a memory management unit (MMU), but some without. The ones most often found in embedded devices are ARM, MIPS, PowerPC, and X86, each in 32- and 64-bit variants, and all of which have memory management units.

Most of this book is written with this class of processor in mind. There is another group that doesn't have an MMU that runs a subset of Linux known as micro controller Linux or uClinux. These processor architectures include ARC, Blackfin, Microblaze, and Nios. I will mention uClinux from time to time but I will not go into details because it is a rather specialized topic.

Secondly, you will need a reasonable amount of RAM. 16 MiB is a good minimum, although it is quite possible to run Linux using half that. It is even possible to run Linux with 4 MiB if you are prepared to go to the trouble of optimizing every part of the system. It may even be possible to get lower, but there comes a point at which it is no longer Linux.

Thirdly, there is non-volatile storage, usually flash memory. 8 MiB is enough for a simple device such as a webcam or a simple router. As with RAM, you can create a workable Linux system with less storage if you really want to but, the lower you go, the harder it becomes. Linux has extensive support for flash storage devices, including raw NOR and NAND flash chips and managed flash in the form of SD cards, eMMC chips, USB flash memory, and so on.

Fourthly, a debug port is very useful, most commonly an RS-232 serial port. It does not have to be fitted on production boards, but makes board bring-up, debugging, and development much easier.

Fifthly, you need some means of loading software when starting from scratch. A few years ago, boards would have been fitted with a JTAG interface for this purpose, but modern SoCs have the ability to load boot code directly from removable media, especially SD and micro SD cards, or serial interfaces such as RS-232 or USB.
In addition to these basics, there are interfaces to the specific bits of hardware your device needs to get its job done. Mainline Linux comes with open source drivers for many thousands of different devices, and there are drivers (of variable quality) from the SoC manufacturer and drivers from the OEMs of third-party chips that may be included in the design, but remember my comments on the commitment and ability of some manufacturers. As a developer of embedded devices, you will find that you spend quite a lot of time evaluating and adapting third-party code, if you have it, or liaising with the manufacturer if you don't. Finally, you will have to write the device support for any interfaces that are unique to the device, or find someone to do it for you.

Hardware used in this book

The worked examples in this book are intended to be generic, but to make them relevant and easy to follow, I have had to choose a specific device as an example. I have used two exemplar devices: the BeagleBone Black and QEMU. The first is a widely-available and cheap development board which can be used in serious embedded hardware. The second is a machine emulator that can be used to create a range of systems that are typical of embedded hardware. It was tempting to use QEMU exclusively, but, like all emulations, it is not quite the same as the real thing. Using a BeagleBone, you have the satisfaction of interacting with real hardware and seeing real LEDs flash. It was also tempting to select a more up-to-date board than the BeagleBone Black, which is several years old now, but I believe that its popularity gives it a degree of longevity and means that it will continue to be available for some years yet.

In any case, I encourage you to try out as many of the examples as you can using either of these two platforms, or indeed any embedded hardware you may have to hand.

The BeagleBone Black

The BeagleBone and the later BeagleBone Black are open hardware designs for a small, credit card sized development board produced by Circuitco LLC. The main repository of information is at www.beagleboard.org. The main points of the specification are:

- TI AM335x 1GHz ARM® Cortex-A8 Sitara SoC
- 512 MiB DDR3 RAM
- 2 or 4 GiB 8-bit eMMC on-board flash storage
- Serial port for debug and development
• MicroSD connector, which can be used as the boot device
• Mini USB OTG client/host port that can also be used to power the board
• Full size USB 2.0 host port
• 10/100 Ethernet port
• HDMI for video and audio output

In addition, there are two 46-pin expansion headers for which there are a great variety of daughter boards, known as capes, which allow you to adapt the board to do many different things. However, you do not need to fit any capes in the examples in this book.

In addition to the board itself, you will need:

• a mini USB to full-size USB cable (supplied with the board) to provide power, unless you have the last item on this list.
• an RS-232 cable that can interface with the 6-pin 3.3 volt TTL level signals provided by the board. The Beagleboard website has links to compatible cables.
• a microSD card and a means of writing to it from your development PC or laptop, which will be needed to load software onto the board.
• an Ethernet cable, as some of the examples require network connectivity.
• optional, but recommended, a 5V power supply capable of delivering 1 A or more.

QEMU
QEMU is a machine emulator. It comes in a number of different flavors, each of which can emulate a processor architecture and a number of boards built using that architecture. For example, we have the following:

• `qemu-system-arm`: ARM
• `qemu-system-mips`: MIPS
• `qemu-system-ppc`: PowerPC
• `qemu-system-x86`: x86 and x86_64
For each architecture, QEMU emulates a range of hardware, which you can see by using the option `-machine help`. Each machine emulates most of the hardware that would normally be found on that board. There are options to link hardware to local resources, such as using a local file for the emulated disk drive. Here is a concrete example:

```
$ qemu-system-arm -machine vexpress-a9 -m 256M -drive file=rootfs.ext4,sd -net nic -net use -kernel zImage -dtb vexpress-v2p-ca9.dtb -append "console=ttyAMA0,115200 root=/dev/mmcblk0" -serial stdio -net nic,model=lan9118 -net tap,ifname=tap0
```

The options used in the preceding command line are:

- `-machine vexpress-a9`: creates an emulation of an ARM Versatile Express development board with a Cortex A-9 processor
- `-m 256M`: populates it with 256 MiB of RAM
- `-drive file=rootfs.ext4,sd`: connect the sd interface to the local file rootfs.ext4 (which contains a filesystem image)
- `-kernel zImage`: loads the Linux kernel from the local file zImage
- `-dtb vexpress-v2p-ca9.dtb`: loads the device tree from the local file vexpress-v2p-ca9.dtb
- `-append "..."`: supplies this string as the kernel command line
- `-serial stdio`: connects the serial port to the terminal that launched QEMU, usually so that you can log on to the emulated machine via the serial console
- `-net nic,model=lan9118`: creates a network interface
- `-net tap,ifname=tap0`: connects the network interface to the virtual network interface tap0

To configure the host side of the network, you need the `tunctl` command from the User Mode Linux (UML) project; on Debian and Ubuntu the package is named `uml-utilities`. You use it to create a virtual network using the following command:

```
$ sudo tunctl -u $(whoami) -t tap0
```

This creates a network interface named `tap0` which is connected to the network controller in the emulated QEMU machine. You configure `tap0` in exactly the same way as any other interface.

All of these options are described in detail in the following chapters. I will be using Versatile Express for most of my examples, but it should be easy to use a different machine or architecture.
Software used in this book
I have used only open source software both for the development tools and the target operating system and applications. I assume that you will be using Linux on your development system. I tested all the host commands using Ubuntu 14.04 and so there is a slight bias towards that particular version, but any modern Linux distribution is likely to work just fine.

Summary
Embedded hardware will continue to get more complex, following the trajectory set by Moore's Law. Linux has the power and the flexibility to make use of hardware in an efficient way.

Linux is just one component of open source software out of the many that you need to create a working product. The fact that the code is freely available means that people and organizations at many different levels can contribute. However, the sheer variety of embedded platforms and the fast pace of development lead to isolated pools of software which are not shared as efficiently as they should be. In many cases, you will become dependent on this software, especially the Linux kernel that is provided by your SoC or Board vendor, and to a lesser extent the toolchain. Some SoC manufacturers are getting better at pushing their changes upstream and the maintenance of these changes is getting easier.

Fortunately, there are some powerful tools that can help you create and maintain the software for your device. For example, Buildroot is ideal for small systems and the Yocto Project for larger ones.

Before I describe these build tools, I will describe the four elements of embedded Linux, which you can apply to all embedded Linux projects, however they are created. The next chapter is all about the first of these, the toolchain, which you need to compile code for your target platform.
Where to buy this book

You can buy Mastering Embedded Linux Programming from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.